TIME BOUNDS FOR SHARED OBJECTS IN PARTIALLY SYNCHRONOUS SYSTEMS

A Thesis

by

JIAQI WANG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2011

Major Subject: Computer Science

TIME BOUNDS FOR SHARED OBJECTS IN PARTIALLY SYNCHRONOUS SYSTEMS

A Thesis

by

JIAQI WANG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Jennifer L. Welch
Committee Members,	Andreas Klappenecker
	Alex Sprintson
Head of Department,	Duncan M. Walker

December 2011

Major Subject: Computer Science

ABSTRACT

Time Bounds for Shared Objects in Partially Synchronous Systems.

(December 2011)

Jiaqi Wang, B.S., Nanjing University

Chair of Advisory Committee: Dr. Jennifer L. Welch

Shared objects are a key component in today's large distributed systems. Linearizability is a popular consistency condition for such shared objects which gives the illusion of sequential execution of operations. The time bound of an operation is the worst-case time complexity from the operation invocation to its response. Some time bounds have been proved for certain operations on linearizable shared objects in partially synchronous systems but there are some gaps between time upper bound and lower bound for each operation. In this work, the goal is to narrow or eliminate the gaps and find optimally fast implementations.

To reach this goal, we prove larger lower bounds and show smaller upper bounds (compared to $2d$ for all operations in previous folklore implementations) by proposing an implementation for a shared object with an arbitrary data type in distributed systems of n processes in which every message delay is bounded within $[d-u, d]$ and the maximum skew between processes' clocks is ϵ.

Considering any operation for which there exist two instances such that individually, each instance is legal but in sequence they are not, we prove a lower bound of $d+\min\{\epsilon, u, d/3\}$, improving from d, and show this bound is tight when $\epsilon < d/3$ and $\epsilon < u$.

Considering any operation for which there exist k instances such that each instance separately is legal and any sequence of them is legal, but the state of the object is different after different sequences, we prove a lower bound of $(1-1/k)u$, improving

from $u/2$, and show this bound is tight when $k = n$.

A pure mutator only modifies the object but does not return anything about the object. A pure accessor does not modify the object. For a pure mutator OP_1 and a pure accessor OP_2, if given a set of instances of OP_1, the state of the object reflects the order in which the instances occur and an instance of OP_2 can detect whether an instance of OP_1 occurs, we prove the sum of the time bound for OP_1 and OP_2 is at least $d + \min\{\epsilon, u, d/3\}$, improving from d. The upper bound is $d + 2\epsilon$ from our implementation.

ACKNOWLEDGMENTS

I would like to thank my committee chair, Dr. Welch, and my committee members, Dr. Klappenecker, and Dr. Sprintson, for their guidance and support throughout the course of this research. I also want to thank Dr. Lee, for her suggestions and reviews throughout the discussions of this research.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience. I also want to extend my gratitude to the NSF, which provided the research funding.

Finally, thanks to my mother and father for their encouragement.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

A. What Is the Problem

1. Shared Objects

Shared objects are a key component in today's large distributed systems. Applications ranging from electronic commerce to social media on hand-held devices require shared data. These applications provide the functions of sharing, updating and also accessing information (stored in shared objects) in message passing systems. These functions are realized via operations on the shared object. Each operation has an invocation and a response. Different from the object which can be accessed by only one process, since the operations on the shared object are executed by multiple processes, the operations can be executed simultaneously. In fact, the executions of operations on a shared object overlap in time. Each object has a sequential specification which determines the legal operations and the legal operation sequences on the object. For each shared object, linearizability defines a desired behavior of operations (including the operations which overlap in time), which follows the sequential specification of the object. We will give more detailed explanation and formal definition for linearizability in the following chapters.

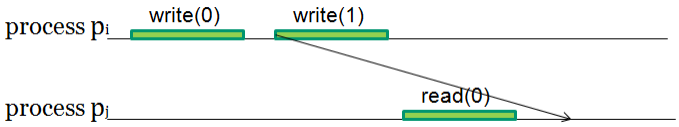
2. Time Bounds

In the applications, the sooner an operation responds, the earlier the application can execute the next step. So we want each operation to respond as fast as possible. Time bound for an operation is the worst-case time complexity of this operation.

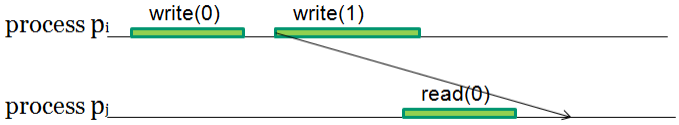
This thesis follows the style of IEEE Transactions on Computers.

The time upper bound for an operation is the minimum value of the worst-case time complexity of this operation in all the existing correct implementations. The time lower bound for an operation is the minimum value of the worst-case time complexity of this operation in all the possible correct implementations. It means in all the correct implementations, for each operation, the worst-case time complexity must reach the lower bound, otherwise the implementation cannot be correct. So if the upper bound equals the lower bound, it means the lower bound is a tight bound and we have found the fastest implementation for this operation; if there exists a gap between the upper bound and lower bound for an operation, either there exists some faster implementation which hasn't been found, or the lower bound is not a tight bound.

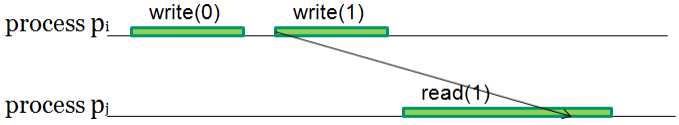
Now we give a simple example to show why the implementation must be incorrect if operations respond faster than their lower bounds. This example is shown on a shared read-write register. Write operation is expressed by $write(arg)$ where arg is the value to be written into the register. Read operation is expressed by $read(ret)$ where ret is the return value of the read operation. In Fig. 1(a), the read operation is invoked after the two write operations complete. By linearizability, the read operation should return 1. However, due to the message delay from process p_i to p_j, the read operation responds before p_j learns the existence of $write(1)$, so the read operation returns 0, violating the linearizability of the register. If we have a longer operation, the problem can be solved. As Fig. 1(b) depicts, if the execution of $write(1)$ is longer such that it overlaps with $read(0)$, then $write(0) \circ read(0) \circ write(1)$ is a legal permutation and it respects the real time sequence. As Fig. 1(c) depicts, if the execution of the read operation is longer, such that p_j learns the existence of $write(1)$ before the read operation responds, then the read operation can return 1 after learning the existence of $write(1)$.



(a) Incorrect Implementation



(b) Longer Write Operation



(c) Longer Read Operation

Fig. 1. Operation Time and Linearizability

3. Problem Statement

From the view of the application, we want the operations to respond faster. But to ensure linearizability of the shared object, the operations cannot respond too fast. So the questions become: without violating the linearizability of the object, how fast can the operations on a shared object respond? And how can one make them respond quickly?

To guarantee the linearizability of an object of arbitrary data type, a centralized mechanism can perform each operation with time at most $2d$ in the worst case if the system provides message delay bound d, since the message from the invoking process to the control center takes at most d and the response message also takes at most d. Alternatively, one can use a total order broadcast primitive, but this is not faster than the centralized scheme when taking into account the time overhead to implement the totally ordered broadcast on top of a point-to-point message system [1]. Increasing pressure to speed up applications raises the question whether operations can be executed faster than $2d$.

B. Previous Related Work

Lipton and Sandberg [5] started to describe fundamental performance limitations for distributed systems that provide sequential consistency (a consistency condition weaker than linearizability). Then Attiya and Welch [1] extended Liption and Sandberg's work to show separation w.r.t. time complexity of operations between sequential consistency and linearizability. They also considered shared objects of type other than read/write registers - specifically, stacks and queues.

To ensure serializability of transactions, Weihl [8] proposed two algorithms using conflict relations based on the commutativity of operations. The two algorithms

permit the results returned by operations to be used in determining conflicts, thus permitting higher levels of concurrency than is otherwise possible.

Inspired by Weihl's work, Kosa [3] proved tight bounds for distributed implementations of different operations types based on the commutativity relationships between operations. Those operations are classified based on their characteristic properties. Kosa [3] used algebraic approaches to specify the properties of operations and defined the operation types such as immediately non-self-commuting operations, immediately self-commuting operations, pure mutators and accessors (definitions of them are given in Chapter II). Then Kosa focused on immediately non-self-commuting operations or pairs of operations which immediately do not commute with each other in a perfectly synchronous system and proved tight time bounds. What's more, Kosa extended the time lower bound proof from a pair of operations to a commutativity graph.

We also study immediately non-self-commuting operations and pairs of operations which immediately do not commute with each other. But we prove the time bounds in a partially synchronous system where the message delay and local clock skew are constrained by certain bounds. And we show that if the local clock skew is smaller than one third of the message delay upper bound, the time bound we proved for immediately non-self-commuting operations is tight. Immediately non-self-commuting operations include read-modify-write on a register, dequeue on a queue and pop on a stack.

Kosa also extended some time lower bounds in partially synchronous systems from previous lower bounds of read and write operations [1]. However the previous time lower bound for the eventually non-self-commuting operations and pure accessors left gaps with respect to their upper bounds. We extend the number of concurrent

---

In a perfectly synchronous system, every process has the same local time.

operations from 2 to an arbitrary positive number k, which means we study permutations of k operations instead of the commutativity of two operations. Then we prove a tight time bound which can be applied to eventually non-self-permuting operations such as write on a register, enqueue on a queue and push on a stack.

There are also some results on pairs of operations in partially synchronous system. Mavronicolas and Roth [7] proved for write and read operations on a shared object, the time lower bound for the sum of them is at least $d + \min\{\epsilon, u\}/2$. However, this result only applies for a certain class of algorithms with some constraint. In this work, we remove the constraint. Also, a more general result instead of only for write and read operations is obtained. In this study, it is found that the time bound for a pair of operations is impacted by the properties of individual operations, such as eventually self-commuting, eventually non-self-commuting, and self-overwriting.

C. What Is Done in the Thesis

This work studies time bounds for linearizable implementations of arbitrary shared objects, with applications to some commonly used shared objects, such as read/write registers, queues, stacks and trees. Each shared object provides a specific semantic structure and is equipped with a specific set of operations. Kosa [3] characterized operations by axioms on what operation sequences are "legal" and proved a variety of upper and lower bounds in different models. But there still exist gaps between the lower and upper bounds for many commonly used data types. In this work, Kosa's approach is extended as follows:

- We derive an algorithm that exploits the axiomatic properties of different operations to reduce the running time of each operation to less than $2d$.
- We obtain better lower bounds on the time complexity of certain types of op-

erations, by exploiting the message delay uncertainty and the presence of more than 2 processes in the system.

- As a result of the new upper and lower bounds, the gap is reduced and in some cases we have tight bounds.
- New classifications of operations are also considered based on new properties relevant to the lower bounds.

Extending Kosa's work, we consider the following properties of operations:

- whether the operation modifies the object;
- whether the operation returns information about the object;
- whether the operation commutes with itself; and
- whether the operation overwrites the whole state of the object.

For example, suppose on a shared register x, there are three operations: read, write and increment.

1. The read operation does not modify the value of x and it returns the value. Besides, the read operation commutes with itself, because in an operation sequence, two consecutive read operations r_1, r_2 return the same value and if we exchange the positions of r_1 and r_2 in an operation sequence, the new operation sequence is still "legal".
2. The write operation modifies the value of x but it does not return any information about x. Besides, the write operation does not eventually commute with itself (refer to Definition C.3 in Chapter II), because if we exchange the positions of two different and consecutive write operations in an operation sequence,

the value of x after the two writes will be different from that before position exchanging. What's more, the write operation overwrites the whole state of x, because whatever the previous operation history is, the latest write operation determines the value of x.

3. In contrast, although the increment operation also modifies x, it commutes with itself and it doesn't overwrite the whole state of x.

Finally we summarize the time bounds on the typical shared object types - register, queue, stack and tree. On these object types there are operations which satisfy the properties we work on. So we can apply lower and upper time bounds on these objects easily.

CHAPTER II

ARBITRARY DATA TYPES

The data type of an object specifies a set of operations which are meaningful to this type. Each operation is composed of an invocation and a response, and the meaning of the operation can be reflected by modifications on the object and the response of the operation. In this work we only consider deterministic objects.

A. Deterministic Object

In this chapter, Chapter III and Chapter IV, we use O to specify an instance of an object, use OP to specify an operation type, such as write operation on a register, and use *op* for an instance of this operation type. We write $op \in OP$ to express that *op* is an operation of type OP. We also use $OP(arg, ret)$ to denote an operation *op* of type OP which takes arg as the operation argument and takes ret as the return value, so $op = OP(arg, ret)$. In Chapter V, we use $op(arg)$ as an operation invocation with argument arg.

For an object which is accessed by a single process, since all the operations on this object are executed sequentially, there is a sequential specification to determine the desired behavior of this object. For each object, the sequential specification contains a set of operations and a set of sequences of operations. The set of sequences of operations contains all the *legal* operation sequences. This set defines whether an operation sequence is legal. For example:

- In the sequential specification of a read-write register, the set of legal sequences of operations contains all the operation sequences in which each read operation returns what the latest write operation before it writes into the register.

- In the sequential specification of a queue, in each legal operation sequence, the dequeue operation removes and returns the head of the queue.
- In the sequential specification of a stack, in each legal operation sequence, the pop operation removes and returns the element at top of the stack.

Definition A.1 *A* deterministic *object satisfies the following condition:*

For every operation sequence ρ and every operation type OP, if both $\rho \circ OP(arg, ret)$ and $\rho \circ OP(arg, ret')$ are legal, then $ret = ret'$.

In the whole work, we only consider deterministic objects.

B. Immediately Non-commuting

Definition B.1 ***Immediately non-commuting*** *- For any two operation types OP_1 and OP_2, if there exist an operation sequence ρ, operation $op_1 \in OP_1$, and operation $op_2 \in OP_2$, such that $\rho \circ op_1$ and $\rho \circ op_2$ are legal but at least one of the below is illegal:*

(1) $\alpha = \rho \circ op_1 \circ op_2$

(2) $\beta = \rho \circ op_2 \circ op_1$

we say OP_1 and OP_2 are immediately non-commuting.

For example, suppose OP_1 is the read operation and OP_2 is the write operation on a shared register. If $\rho = write(0)$, then $\rho \circ write(1)$, $\rho \circ read(0)$ and $\rho \circ\ read(0) \circ write(1)$ are all legal, but $\rho \circ write(1) \circ read(0)$ is not legal. Thus, the read and write operation are immediately non-commuting operations.

Definition B.2 ***Immediately non-self-commuting*** *- In the definition of immediately non-commuting, if $OP_1 = OP_2$, then OP_1 is immediately non-self-commuting.*

Notes: in the following chapters we describe two operation types which are not immediately non-commuting by immediately commuting and describe an operation type which is not immediately non-self-commuting by immediately self-commuting.

Definition B.3 ***Strongly Immediately non-self-commuting*** *- In the definition of immediately non-commuting, if both α and β are illegal and $OP_1 = OP_2$, then OP_1 is strongly immediately non-self-commuting.*

Many common immediately non-self-commuting operation types are also strongly immediately non-self-commuting operation types. First we show three commonly used operation types which are both immediately non-self-commuting and also strongly immediately non-self-commuting.

- Read-Modify-Write on a register R: Suppose $\rho = write(0)$, op_1 reads the value of R and writes 1 into R and op_2 reads the value of R and writes 2 into R. Because both $\rho \circ op_1$ and $\rho \circ op_2$ are legal, both op_1 and op_2 should return 0. But it is illegal if op_2 returns 0 in $\rho \circ op_1 \circ op_2$ and it is illegal if op_1 returns 0 in $\rho \circ op_2 \circ op_1$.

- Pop on a stack S: Suppose after ρ, S has one element X. op_1 and op_2 pop the top of S and return the element they pop. Because $\rho \circ op_1$ and $\rho \circ op_2$ are legal, both op_1 and op_2 should return X. Then both $\rho \circ op_1 \circ op_2$ and $\rho \circ op_2 \circ op_1$ are illegal, because op_2 should return nothing in $\rho \circ op_1 \circ op_2$ and op_1 should return nothing in $\rho \circ op_2 \circ op_1$.

- Dequeue on a queue Q: Suppose after ρ, Q has one element X. op_1 and op_2 are both dequeue operations which remove the head of Q and return the element they remove. Because $\rho \circ op_1$ and $\rho \circ op_2$ are legal, both op_1 and op_2 return X.

Then both $\rho \circ op_1 \circ op_2$ and $\rho \circ op_2 \circ op_1$ are illegal, because op_2 returns nothing if $\rho \circ op_1 \circ op_2$ is legal and op_1 returns nothing if $\rho \circ op_2 \circ op_1$ is legal.

Next we describe an operation type $UpdateNext(i, a, b)$ on an integer array of size 2, which is immediately non-self-commuting but not strongly immediately non-self-commuting. $UpdateNext(i, a, b)$ returns the ith element in the array (a shows the return value) and updates the $(i+1)$th element with value b; and if the ith element is the last one in the array, it modifies nothing.

To show it is immediately non-self-commuting, suppose the array is initialized with $[x, y]$ and previous operation sequence ρ is empty. $op_1 = OP(1, x, z)$ where $z \neq y$, $op_2 = OP(2, y, z)$. Then $\rho \circ op_1$, $\rho \circ op_2$ and $\rho \circ op_2 \circ op_1$ are legal. But $\rho \circ op_1 \circ op_2$ is illegal.

Now we show $UpdateNext(i, a, b)$ is not a strongly immediately non-self-commuting operation type. Suppose in contradiction we can find an operation sequence ρ, $op_1 = OP(i, a, b)$, $op_2 = OP(j, a', b')$ such that both $\rho \circ op_1 \circ op_2$ and $\rho \circ op_2 \circ op_1$ are illegal. Without loss of generality, we can assume after ρ, the array value is $[x, y]$. We show that in all the cases below, there exists some contradiction:

- Case 1: $i = j = 1$. Because $\rho \circ op_1$ and $\rho \circ op_2$ are legal, $a = a' = x$. Then both $\rho \circ op_1 \circ op_2$ and $\rho \circ op_2 \circ op_1$ are legal, because no operation changes the first element.

- Case 2: $i = j = 2$. Similar to case 1, both $\rho \circ op_1 \circ op_2$ and $\rho \circ op_2 \circ op_1$ are legal.

- Case 3: $i = 1, j = 2$. Because $\rho \circ op_1$ and $\rho \circ op_2$ are legal, $a = x$ and $a' = y$. Then $\rho \circ op_2 \circ op_1$ must be legal, because op_2 modifies nothing.

- Case 4: $i = 2, j = 1$. Similar to case 3, $\rho \circ op_1 \circ op_2$ is legal.

C. Eventually Non-commuting and Eventually Non-permuting

In contrast with immediately non-self-commuting operation types, there exist eventually non-self-commuting operation types. Before we give the definition for them, let's review Kosa [3]'s definitions for "look like" and "equivalent" first.

Definition C.1 ***Look like*** - *An operation sequence ρ_1 looks like another operation sequence ρ_2, if for any operation sequence ρ_3 where $\rho_1 \circ \rho_3$ is legal, $\rho_2 \circ \rho_3$ is legal.*

Definition C.2 ***Equivalent*** - *Two operation sequences ρ_1 and ρ_2 are equivalent if and only if ρ_1 looks like ρ_2 and ρ_2 looks like ρ_1.*

Definition C.3 ***Eventually non-self-commuting*** - *For any eventually non-self-commuting operation type OP, there exists an operation sequence ρ, and $op_1, op_2 \in OP$, such that $\rho \circ op_1$ and $\rho \circ op_2$ are legal, and $\rho \circ op_1 \circ op_2$ and $\rho \circ op_2 \circ op_1$ are not equivalent.*

An example is the write operation on a read/write register. Suppose after $\rho = write(0)$. op_1 writes 1 into the register and op_2 writes 2 into the register. After $\rho \circ op_1 \circ op_2$, the value of the register is 2. After $\rho \circ op_2 \circ op_1$, the value of the register is 1. Therefore $\rho \circ op_1 \circ op_2$ does not look like $\rho \circ op_2 \circ op_1$ because there exists an operation $read(1)$ such that $\rho \circ op_2 \circ op_1 \circ read(1)$ is legal while $\rho \circ op_1 \circ op_2 \circ read(1)$ is illegal.

Now we have defined immediately non-commuting and eventually non-commuting. We can see that both of them are defined based on the order of 2 operations. However, if we want to implement a shared object in a message passing system, the message passing system usually has more than 2 processes and it is reasonable that more operations are invoked concurrently. Thus we consider the order of $n (n \geq 2)$ overlapping

operations. We call an operation sequence composed of the n operations a permutation of the n operations. So we use permuting to describe relationships between the n operations. Similar to eventually non-self-commuting, the property of eventually non-self-permuting is determined by if two different permutations of $n(n \geq 2)$ operations are equivalent or not.

Before we give the definition of non-self-permuting, we continue comparing commuting and permuting . If there are only 2 operations, there are only 2 permutations of operation sequence, depending on which operation is the first one. So either the two permutations are equivalent or not equivalent. If there are n operations where $n > 2$, there are $n!$ permutations, which means in the $n!$ permutations, some permutations may be equivalent while some other permutations may be different. An extreme case is that any two different permutations of the n operations are not equivalent. On the opposite side, another extreme case is that all the different permutations are equivalent. So between the strong non-permuting (the first extreme case) and permuting (the second extreme case), there might be different levels of the non-permuting properties.

Now let's start the definitions related to the non-permuting property.

Definition C.4 ***Eventually non-self-any-permuting*** - *An operation type OP is eventually non-self-any-permuting if and only if there exist an operation sequence ρ, and $op_1 \in OP, op_2 \in OP, ...op_n \in OP$ for any $n > 1$, such that:*

1. *$\rho \circ op_i (0 < i \leq n)$ is legal;*
2. *there exist at least two legal permutations of the n operations; and*
3. *any two different legal permutations of the n operations are not equivalent.*

Definition C.5 ***Eventually non-self-last-permuting*** - *An operation type OP is*

eventually non-self-last-permuting if and only if there exist an operation sequence ρ*, and* $op_1 \in OP, op_2 \in OP, ...op_n \in OP$ *for any* $n > 1$*, such that:*

1. $\rho \circ op_i (0 < i \leq n)$ *is legal;*
2. *there exist at least two legal permutations of the* n *operations; and*
3. *any two different legal permutations* π*,* π' *of the* n *operations are not equivalent if* $last(\pi) \neq last(\pi')$ *(*$last(\pi)$ *is the last operation in* π*).*

Of course the eventually non-self-any-permuting operations are eventually non-self-last-permuting operations, because two different permutations with different last operations belong to two different permutations. Let's see some examples below:

- Write: The write operation on a read/write register is an eventually non-self-last-permuting operation but not an eventually non-self-any-permuting operation. Because if $last(\pi) = last(\pi')$, then $\rho \circ \pi$ and $\rho \circ \pi'$ are equivalent. If $last(\pi) \neq last(\pi')$, they are not equivalent. So for any two permutations, as long as the last operation is the same, they are equivalent, which means write operation is not eventually non-self-any-permuting.

- Push: The push operation on a stack is eventually non-self-any-permuting operations. Suppose the stack is initialized with an empty stack, ρ is empty, and let op_i push the integer i onto the stack. For any two different permutations π and π', $\rho \circ \pi$ and $\rho \circ \pi'$ are not equivalent because a sequence of continuous pop operations can distinguish the differences between the two permutations.

- Enqueue: The enqueue operation on a queue is eventually non-self-any-permuting operations. Suppose the queue is initialized with an empty queue, ρ is empty, and let op_i enqueues the integer i into the queue. For any two different permutation π and π, $\rho \circ \pi$ and $\rho \circ \pi'$ are not equivalent because a sequence of

continuous dequeue operations can distinguish the differences between the two permutations.

Definition C.6 ***Eventually self-commuting*** - *An operation type OP is eventually self-commuting if and only if for any operation sequence ρ, and any $op_1, op_2 \in OP$ such that $\rho \circ op_1$ and $\rho \circ op_2$ are legal, $\rho \circ op_1 \circ op_2$ and $\rho \circ op_2 \circ op_1$ are legal and equivalent.*

For example, consider the insert and delete operations on a set. The order of insertion or deletion does not affect the elements in the set, so they are eventually self-commuting.

D. Mutator, Accessor and Overwriter

Definition D.1 ***Mutator*** - *An operation which modifies the object. For any mutator OP, there exist an operation sequence ρ and operation $op_1 \in OP$, such that $\rho \circ op_1$ and ρ are not equivalent.*

Definition D.2 ***Accessor*** - *An operation which return some information about the object. For any accessor OP, there exist a legal operation sequence ρ and operation instance $op \in OP$, such that $\rho \circ op$ is illegal.*

Definition D.3 ***Pure Mutator*** - *Mutator which is not accessor.*

Definition D.4 ***Pure Accessor*** - *Accessor which is not mutator.*

Definition D.5 ***Non-overwriter*** - *A mutator OP is a non-overwriter if there exist operation sequences ρ, and $op_1, op_2 \in OP$ such that $\rho \circ op_1 \circ op_2$ and $\rho \circ op_2$ are not equivalent.*

For example, suppose op_1 and op_2 are increment operations which return nothing on a shared register and op_3 is the read operation on this register. Suppose $\rho = write(0)$, op_1 increases the value by 1, op_2 increases the value by 2 and op_3 is a read operation. Then if $\rho \circ op_1 \circ op_2 \circ op_3$ is legal, op_3 should return 3 and if $\rho \circ op_2 \circ op_3$ is legal, then op_3 should return 2. Therefore, one of them is illegal and increment is a non-overwriter.

CHAPTER III

SYSTEM MODEL

A. Overview

The shared object system contains three layers: (1) application layer; (2) object implementation layer; and (3) message passing layer. Each layer is built on another layer as shown in Fig. 2:

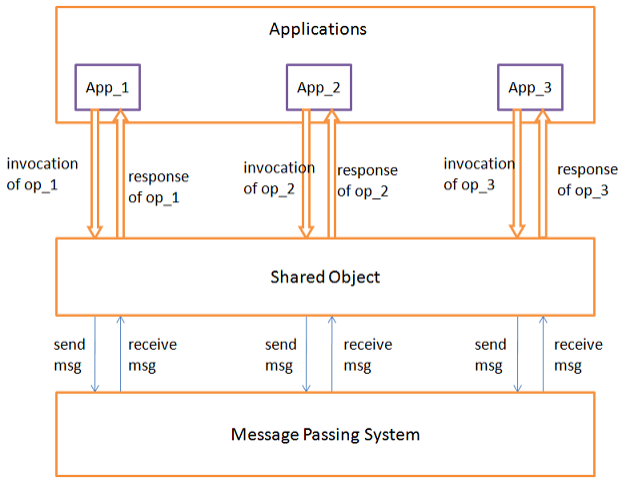


Fig. 2. Shared Object Implementation in Message Passing System

The top layer is the application layer, which sends operation invocations to the middle layer and receives the operation responses from the middle layer. Each operation must be invoked and respond in the same process. Each process can have at most one pending operation, which means only after returning the response of the previous operation, a process can invoke a new operation. Besides, in a finite time period, every process can invoke only a finite number of operations.

The middle layer is the implementation of the shared object, which takes oper-

ation invocations from the application layer as input and takes operation responses as output. It sends and receives messages via the base layer - the message passing system.

The base layer is the message passing system, which takes the message sending events as input and takes message receiving events as output. The message passing system provides communication between processes via reliable message deliveries, which can guarantee: a) every received message must have been sent by some process before; b) every message is received by the destination process eventually; c) every message is received by the destination process only once.

B. Detailed Specifications

1. Process

Each *process* is modeled as a state machine with a set of states and a transition function. The state set has a subset of initial states. The transition function takes as inputs:

- current state
- an input event (operation invocation or message receipt or timer going off), and
- clock time

and produces as outputs:

- new state and
- set of output events (at most one operation response, and for each other process, at most one message to be sent).

Timers are part of the state: each *timer* is a special kind of variable that holds either $\perp$ or a real number that indicates the clock time when the process should take a step. If the input event is a timer going off, then there must be at least one timer in the current state that is equal to the clock time. In the new state, each timer must be set either to $\perp$ or to some value that is larger than the clock time.

A state is *quiescent* if no timer is set. Thus the process will not do anything until it receives a message or gets an operation invocation.

A *step* of a process is a quintuple <current state, input event, clock time, new state, set of output events>, which captures the application of the transition function. We denote the clock time of step σ_i by $clock\_time(\sigma_i)$.

2. View of Process

A *view* of process p_j is a (finite or infinite) sequence $\sigma_1, \sigma_2, \ldots$ of steps of p_j such that

- in σ_1, the current state is quiescent;
- the current state in σ_i equals the new state in σ_{i-1} for all $i > 1$;
- if σ_i includes an operation invocation, then no operation is currently pending at p_j (the constraint on the application);
- the clock times are increasing, and for any finite number x, there are only a finite number of steps whose clock times are less than x; and
- the clock time of σ_i is less than or equal to every timer in the current state for all i.

A *complete view* is a view which either is infinite or ends in a quiescent state. As a result of the last two conditions for a view, if a timer remains set, then a step must

happen at that clock time. For modeling convenience, we assume each message sent has a unique id that also indicates the sender and the recipient.

A *timed view* of process p_j is a view of p_j together with a nonnegative real number t_i associated with each step σ_i in the view. This is the real time at which the step σ_i occurs and is denoted by $real\_time(\sigma_i)$. The following constraint must hold: There exists a constant c_j such that, for all i, $clock\_time(\sigma_i) = real\_time(\sigma_i) + c_j$. That is, we are only considering clocks that run at the same rate as real time. We call this transition function from real time to clock time the clock function for p_j.

$shift(V, x)$ is a shifted timed view, where V is a timed view and x is a real number, by which the real time of each step in V is added.

Claim B.1 *If V is a timed view and x is a real number, then $shift(V, x)$ is a timed view.*

A timed view V_2 is an *extension* of another timed view V_1 if V_1 is a prefix of V_2.

3. Run

A *run* is a set of timed views, one for each process in the set Π of n processes, such that every message received in the run is sent exactly once in the run. Note here a message sent is not necessarily received in the run. A *complete run* is a run in which every timed view in the run is complete and every message is delivered. A *finite run* is a run in which every timed view is finite.

A run R is *admissible* if:

- every message received in R is sent Δt $(d - u \leq \Delta t \leq d)$ real time earlier in R;
- for each message m which is sent at real time t in R, but not received in R, the view of m's recipient in R ends before real time $t + d$; and

- for any two processes p_j and p_k, $|c_j - c_k| \leq \epsilon$ (recall that p_j's clock is offset from real time by c_j and p_k's clock is offset from real time by c_k) for fixed ϵ (i.e., the clock skew is bounded).

A run R_2 is an *extension* of run R_1 if, for each process p_i in Π, the timed view for p_i in R_2 is an extension of the timed view for p_i in R_1.

Claim B.2 *If R is a finite admissible run, then there exists a complete admissible run that is an extension of R.*

Given a run $R = \{V_i : p_i \in \Pi\}$, where each V_i is a timed view of p_i, and a vector $\vec{x}$ of n real numbers, then $shift(R, \vec{x}) = \{shift(V_i, x_i) : p_i \in \Pi\}$.

Claim B.3 *If R is a run and $\vec{x}$ is a vector of real numbers, then $shift(R, \vec{x})$ is a run. Furthermore, the shift operator preserves the property of being complete and the property of being finite, but does not necessarily preserve the property of being admissible.*

A run R_2 can be *appended* to another run R_1 by appending each timed view in R_2 to the corresponding timed view in R_1. We say R_2 is *appendable* to R_1 if:

1. R_1 ends with every process being quiescent (so each timed view is finite); and
2. for each process
   - last state in its timed view in R_1 equals the first state in its timed view in R_2,
   - the first clock time in its timed view in R_2 is greater than the last clock time in its timed view in R_1, and
   - the clock function for the process is the same in R_1 and R_2.

Claim B.4 *Based on the definition of a run, the result of this appending operation is a run.*

A run is *initialized* if processes start in initial states corresponding to the initial value of the object.

4. Correctness of Algorithm

For a shared object, since operation executions overlap in time, there must be some rule to specify the desired behavior or correctness for the overlapping operation. Linearizability [2] (or atomicity [4]) is a popular and easy-to-use consistency condition for such shared objects which gives the illusion of sequential execution of operations.

Linearizability: There exists a permutation π of all the operations in each complete admissible run R such that:

- a) for each object O, the restriction of π to operations on the object O is legal;
- b) if the response of op_1 occurs before the invocation of op_2 in R, then op_1 appears before op_2 in π.

The first criteria follows the sequential specification to determine if the operation sequence π is legal. The second one respects the real time sequence. Two operations that overlap in R may appear in π in either order, but for non-overlapping operations, the operation which completes earlier must appear in π before the one which is invoked later.

A distributed algorithm is correct if, in every complete admissible run R of the algorithm:

- Every operation invocation has a matching response, and every response has a matching invocation (i.e., every operation completes and invocations and responses are properly interleaved).

- Linearizbility of the object is not violated.

We make the following three assumptions about the algorithms we consider, somewhat restricting the class of algorithms for which our lower bounds apply.

1. *Bounded-Time Operations:* There exists a bound B_{op} such that every operation invoked in every complete admissible run has its response within B_{op} time.

2. *Bounded Quiescence:* There exists a bound B_q such that for every complete admissible run containing a finite number of operation invocations, the run is finite and every process is quiescent within B_q real time after the last operation response occurs in the run. Besides, each process remains quiescent until a new operation is invoked or a new message is received.

3. *History-Obliviousness:* Let R_1 and R_2 be two complete admissible runs such that for some process p_i, the same finite sequence of operations is executed by p_i in both R_1 and R_2, and the other processes do not execute any operations in R_1 or R_2. Because of bounded-time operations and bounded quiescence, R_1 and R_2 are both finite. Then for each process p_j, the final state of p_j is the same in both R_1 and R_2. I.e., nothing about the clock times at which events occurred, or the order in which messages arrived is recorded permanently in the states; all that matters is the sequence of operations executed.

CHAPTER IV

LOWER BOUNDS FOR LINEARIZABLE SHARED OBJECT

Three factors may impact the time bound of operations: message delay upper bound d, message delay uncertainty u and local clock skew bound ϵ. We already know the optimal ϵ is smaller than u.

In this chapter, we denote the maximum of the worst-case time from the invocation of each operation $op \in OP$ to the response of op in the fastest implementation by $|OP|$.

Before we analyze the time lower bound of operations, we describe a broadly used technique – standard time shift and introduce a new technique – modified time shift, which helps us prove larger lower bounds.

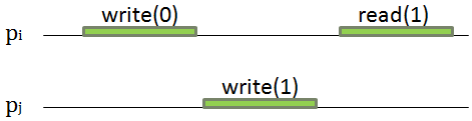
A. Standard Time Shift

We consider a run in which messages between every two processes p_i and p_j have fixed time delay, denoted by $d_{i,j}$. In the standard time shift, the local clock of each process p_i is shifted by certain amount $clock\_shift(i)$. Then in the run resulting from the standard shift, the new message delay $d'_{i,j}$ is [6]:

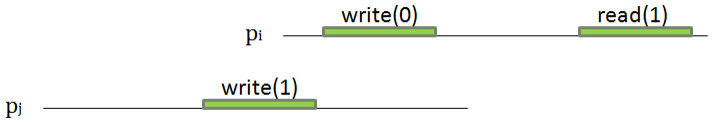
$$d'_{i,j} = d_{i,j} - clock\_shift(i) + clock\_shift(j) \tag{4.1}$$

With the new message delays, each message is still received at the same local time, which means no process notices any difference from before the standard time shift. Therefore the executions of operations are exactly the same as before the standard time shift.

If the original delays and the shift amounts are such that the old and new delays are both admissible, then certain deductions can be made to show some lower bounds.



(a) Before Time Shift



(b) After Time Shift

Fig. 3. Example of Standard Time Shift

Now we show a simple example of a proof with standard time shift [1]. Suppose we want to prove that the lower bound for the write operation is x. Assume in contradiction, there exists a correct implementation of a read-write register in which the worst-case time complexity of the write operation is less than x. First we let two processes p_i and p_j execute operations as Fig. 3(a). In this run, $write(1)$ is invoked immediately after $write(0)$ completes. For this run, operation sequence $write(0) \circ write(1) \circ read(1)$ is legal and respects the real time sequence. Then we use the standard time shift to shift the local clock in process p_i by $2x$ and all the message delays are still admissible after the standard shift, as shown in Fig. 3(b). Because the executions of operations are exactly the same as before the standard time shift, the read operation still returns 1. But after the standard time shift, there is no operation sequence of the three operations which is legal and respects the real time sequence, because $write(0)$ is invoked after $write(1)$ completes. Therefore, the assumed correct implementation does not exist.

In the example above, to prove a lower bound x, we shift the local clock of p_i by $2x$. On one side, we can imagine that by shifting with larger amounts, we can prove a larger lower bound. On the other side, the new message delays must fall into the range $[d-u, d]$ and they are determined by the original message delays and shift amounts, meaning that the shift amounts are limited, otherwise the new message delays are invalid. For instance, in the previous example, by setting $x > u/2$, the new message delays are increased or decreased from old message delays by more than u. Since the old message delays are within $[d-u, d]$, the new message delays must be invalid. To overcome the limitation of shift amounts, we develop the modified time shift in the following section.

B. Modified Time Shift

We develop the modified time shift from the standard one. The intuition is to prove larger lower bounds by shifting the local clocks with larger amounts and the general idea is as below.

In the standard time shift, all the messages delays are still valid in the shifted run. In the modified shift of a run, depending on the original message delays and the time shift amounts, the new message delays may be invalid. The weaker guarantee allows us to shift by larger amounts and thus prove larger lower bounds than before. But we must get rid of the invalid messages. We chop the shifted run to get a prefix of it, such that the invalid messages are not received in the prefix and so the prefix is admissible. After the the run is chopped, there is no invalid message, but the run is not a complete run. So we need to extend the chopped run to a complete run.

With the three steps (1) modified time shift, (2) chopping, and (3) extending, we get a new run, whose prefix is the same as the original run. Therefore if an operation

is invoked and responds within the prefix, the operation returns the same value as in the original run. In this way, we can make use of the modified time shift in a similar way as the standard time shift.

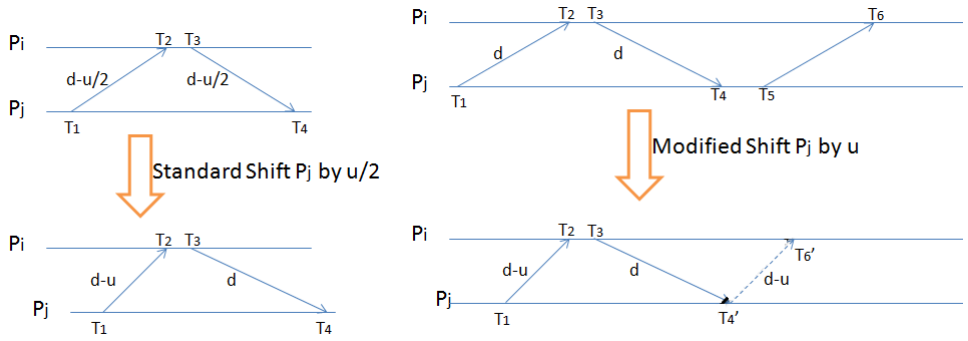
Now we illustrate the differences between standard time shift and modified time shift, and the three steps by simple examples in Fig. 4.

Part (a) in Fig. 4 is an example of the standard time shift. Before the time shift, $d_{i,j} = d_{j,i} = d - u/2$. $clock\_shift(i) = 0$ and $clock\_shift(j) = u/2$. According to formula (1), after the standard time shift, $d'_{i,j} = d - u/2 - 0 + u/2 = d$, and $d'_{j,i} = d - u/2 - u/2 + 0 = d - u$, both of which are within the range $[d - u, d]$ and admissible.

Part (b) in Fig. 4 is an example of the modified time shift. Different from part (a), $d_{i,j} = d_{j,i} = d$.

- Step 1: We shift the local clocks in processes p_i and p_j with $clock\_shift(i) = 0$ and $clock\_shift(j) = u$. According to formula (1), $d'_{i,j}$ is $d + u$, which is not admissible and $d'_{j,i}$ is $d - u$.

- Step 2: To get an admissible prefix of the shifted run, we consider the view in each process: there are two messages between p_i and p_j. The first message is sent from p_j to p_i and does not violate message delay constraints. The second message is sent from p_i to p_j and is not admissible. For process p_j, we chop the view so that the prefix ends before the clock time T'_4, because the invalid message is not received before T'_4. In process p_i, since the view of p_j is chopped to end before T'_4 and the message sent at T'_4 costs at least $d - u$ to be received by p_i (clock time T'_6), the view of p_i is chopped to end before T'_6; otherwise the message which is sent at T'_4 from p_j to p_i is sent after the prefix but received in the prefix.



(a) Standard Time Shift (b) Modified Time Shift

Fig. 4. Standard Time Shift and Modified Time Shift

- Step 3: Extend the prefix to a complete run by letting the second message be received at T_4' (message delivery costs d, which is admissible). Then all the messages are received and the run is complete.

1. Chopping and Extending

Let R be a run with pairwise uniform message delays $\{d_{k1,k2} : p_{k1} \in \Pi, p_{k2} \in \Pi\}$ such that the maximum clock skew is bounded by ϵ and exactly one message delay $d_{i,j}$ is invalid. Let m be the first message sent from p_i to p_j in R, say at real time t_s. Let $t^* = t_s + \min\{d_{i,j}, \delta\}$ (δ is a parameter in the range $[d-u, d]$). Let V_j be the prefix of p_j's timed view in R that ends just before real time t^* (the cross sign in Fig. 5). For each $k \neq j$, let V_k be the prefix of p_k's timed view in R that ends just before real time $t^* + D_{j,k}$, where $D_{j,k}$ is the shortest path distance from p_j to p_k in the complete directed graph whose node set is Π and in which the edge from p_{k1} to p_{k2} is weighted with $d_{k1,k2}$ for all $p_{k1}, p_{k2} \in \Pi$. Then $chop(R, \delta)$ is defined to be $\{V_i : p_i \in \Pi\}$.

Lemma B.1 *If R is a run with pairwise uniform message delays that satisfies admissibility except that exactly one message delay is invalid, then $R' = \text{chop}(R, \delta)$ is*

an admissible run for all $\delta \in [d-u, d]$.

PROOF. First, R' is a run, because

(1) every element in R' is chopped from a timed view in R, which means it is also a timed view; and

(2)all the messages received in R' are sent in R'. Suppose in contradiction, there is a message m' sent from p_{k_1} to p_{k_2}, which is sent after $t^* + D_{j,k_1}$ and received before $t^* + D_{j,k_2}$. Then $D_{j,k_2} - D_{j,k_1} > d_{k_1,k_2}$. So $D_{j,k_1} + d_{k_1,k_2} < D_{j,k_2}$ and D_{j,k_2} is not the shortest path distance from p_j to p_{k_2}.

Second, R' is admissible, because:

1. All the messages received in R' have delays in the range $[d-u, d]$. This is because there is only one invalid message delay $d_{i,j}$ and we cut the view V_j to end before the invalid message is received.

2. For each message which is sent at real time t in R' but not received in R', the view of the recipient is chopped before $t + d$. For the message m from p_i to p_j, because $t^* \leq t_s + \delta \leq t_s + d$, p_j ends before $t_s + d$. For all other messages, because their message delays are no larger than d, if they are not received in R', the recipient ends before the sending time plus d.

3. R' has the same clock functions as R, for which the maximum clock skew is no larger than ϵ.

□

Remark: $t^* = t_s + \min\{d_{i,j}, \delta\}(d-u \leq \delta \leq d)$ ensures $t^* \leq t_s + \delta$, which means V_j ends before $t_s + \delta$ in R'. When R' is extended, we can let the message delay from p_i to p_j fall into the range $[\delta, d]$. Since m is sent at real time t_s, m is received after

$t_s + \delta$ and so after V_j ends. Then m still does not affect R'. And since the message delay is admissible within $[\delta, d]$, we get an admissible extension of R'.

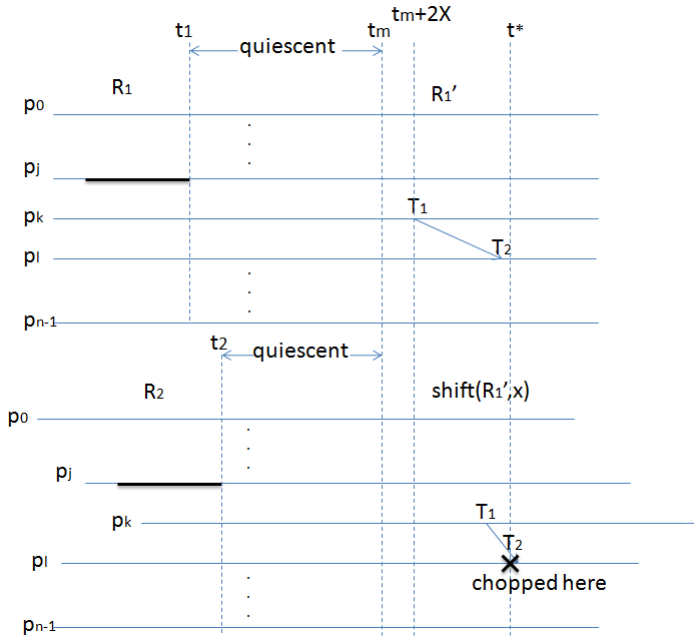


Fig. 5. Chopping and Appending to an Initialized Run

2. Appending to an Initialized Run

If the chopped run is not initialized, there might be a problem about whether it is appendable to an initialized run. To answer this question, we have two claims below.

Given a legal sequence ρ of finite operations on a shared object, define $\mathscr{R}_\rho$ to be the set of all runs R satisfying the following conditions:

- R is an initialized, complete and admissible run;
- process p_0 executes operations in ρ sequentially, and no other process executes any operation;

- the first operation is invoked at real time t_0; and
- each operation (except the first one) is invoked immediately after the previous operation responds.

The bounded time operation and bounded quiescence properties (from Chapter III) show the following claim:

Claim B.2 *For every $R \in \mathscr{R}$, R is finite and every process is quiescent by real time $t_q = t_0 + |\rho| \times B_{op} + B_q$.*

Let R_1 be any run in $\mathscr{R}$. Let R_1' be an admissible run which is appendable to R_1 and starts at real time $t_q + 2X$, where X is a real number which bounds the shift amounts in the following claim (Fig. 5).

Claim B.3 *For every n-vector $\vec{x}$ with $|x_i| \leq X, 0 \leq i \leq n-1$, there exists a run $R_2 \in \mathscr{R}$ such that $shift(R_1', \vec{x})$ is appendable to R_2.*

PROOF. Let R_2 have clock functions shifted from those of R_1 by $\vec{x}$. In R_2, let all processes start with their initial states and let p_0 invoke the operations in ρ starting at real time t_0 without any time gap between two operations. Then R_2 is one element of $\mathscr{R}$. $shift(R_1', \vec{x})$ is appendable to R_2, because:

1. According to the definition of appendable runs, for each process p_i, the clock functions in R_1' are the same as the clock functions in R_1. Because clock times in R_2 are shifted by $\vec{x}$ from R_1, $shift(R_1', \vec{x})$ has the same clock functions as in R_2.
2. Because R_1' starts after $t_q + 2X$, $shift(R_1', \vec{x})$ starts after $t_q + X$. So for each process, the first clock time in $shift(R_1', \vec{x})$ is later than the last clock time in R_2.

3. Based on the assumption of History-Obliviousness, every process p_i ends with the same state in R_1 and R_2, which is the same as the first state in R_1' and in $shift(R_1', \vec{x})$.

□

Corollary B.4 *If there is only one invalid message delay in* $shift(R_1', \vec{x})$, $|x_i| \leq X$, R_1' *is appendable to the run* $R_1 \in \mathscr{R}_\rho$ *and starts after* $t_q + X$, *then there exists a run* $R_2 \in \mathscr{R}_\rho$, *such that* $chop(shift(R_1', \vec{x}), \delta)$ *is appendable to* R_2.

Note: The assumption of "Bounded Operation", "Bounded Quiescence" and "History Obliviousness" are used in Claim B.3. This claim is used in the following theorems to exclude the impact of previous operation history. The operation response should depend on the initialization of the object, sequential operation history and other concurrent operations. The details of previous operations such as execution times should be unrelated. Here we consider the objects with initialization constraints that the object can only be initialized with certain specific values. If there is no constraint in the initialization, we can initialize the object with the state after the previous operation sequence. In this way, the impact of the previous operation execution details has already been excluded when the object is initialized.

C. Strongly Immediately Non-self-commuting Operation Types

Theorem C.1 *For every strongly immediately non-self-commuting operation type* OP, $|OP| \geq d + m$ *time* ($m = \min\{\epsilon, u, d/3\}$) *for any linearizable shared object implemented in an* n*-process*($n \geq 3$) *message passing system with message delay bound* $[d - u, d]$ *and* ϵ*-bounded clocks.*

Proof. Suppose, in contradiction that on a linearizable shared object, there exists

an operation type OP which is strongly immediately non-self-commuting, but the operations of this type cost less than $d + m$ time. From the definition of strongly immediately non-self-commuting, there exist an operation sequence ρ on this object, operations $op_1 \in OP$ and $op_2 \in OP$, such that

(1) $\rho \circ op_1$ is legal

(2) $\rho \circ op_2$ is legal

(3) $\rho \circ op_1 \circ op_2$ is illegal

(4) $\rho \circ op_2 \circ op_1$ is illegal

We assume that $op_1 = OP(arg_1, ret_1)$ and $op_2 = OP(arg_2, ret_2)$.

The counter-example is constructed with the following 4 steps (Proof structure refers to Figure 6). In the statements of runs in each step, we ignore the operation sequence ρ, assuming the operations in each run are invoked after all operations in ρ have responded and every process is quiescent for a long time.

Step 1: Let p_i and p_j be two specific processes and p_k denotes any other process. In R_1:

- All the processes except process p_j have the same local clock function; p_j's local clock function is m later than the other processes;
- The message delays are pairwise uniform with $d_{i,k} = d_{i,j} = d_{j,i} = d_{k,j} = d$; and with $d_{k,i} = d_{j,k} = d - m$. The message delay from p_k to $p_{k'}$ for any $k \neq i, j$ and $k' \neq i, j$ is any valid delay Δt where $d - u \leq \Delta t \leq d$ (Fig. 7 (a)).

In run R_1, p_i executes the operation $op'_1 \in OP$ with argument arg_1 invoked at real time t (local time T). Then process p_j executes operation $op'_2 \in OP$ with argument arg_2 invoked at real time $t + m$ (local time T). No other operations are invoked.

In run R'_1, every process has the same local clock as in R_1 and the message delays

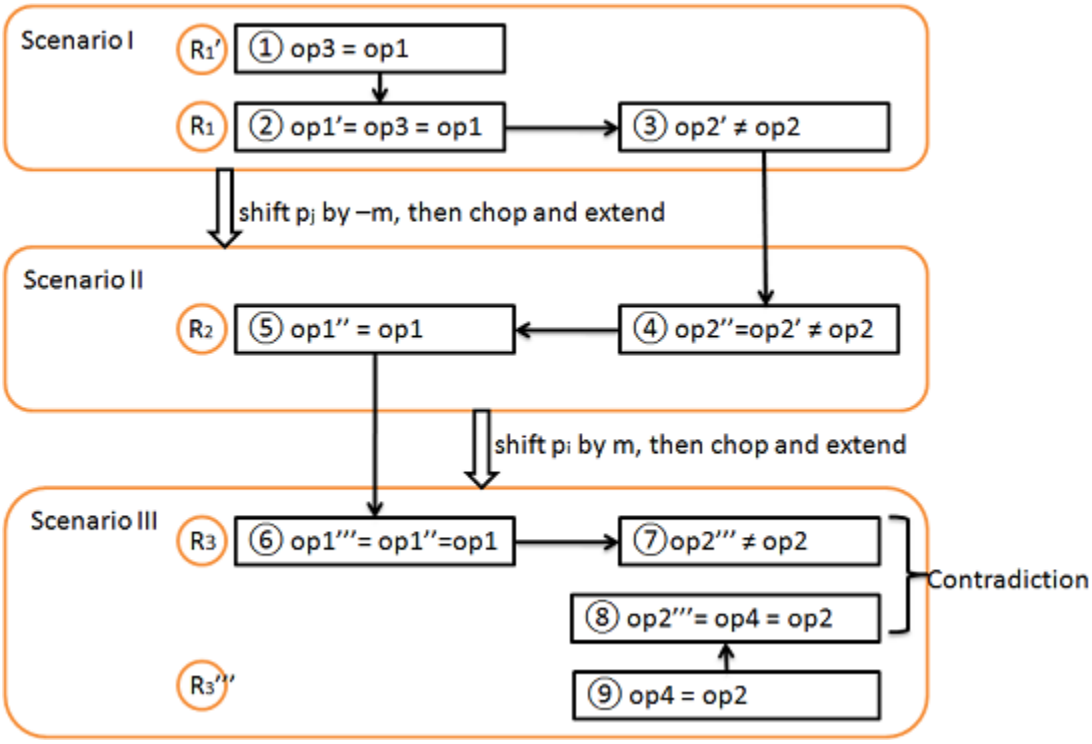


Fig. 6. Proof Structure of Theorem C.1

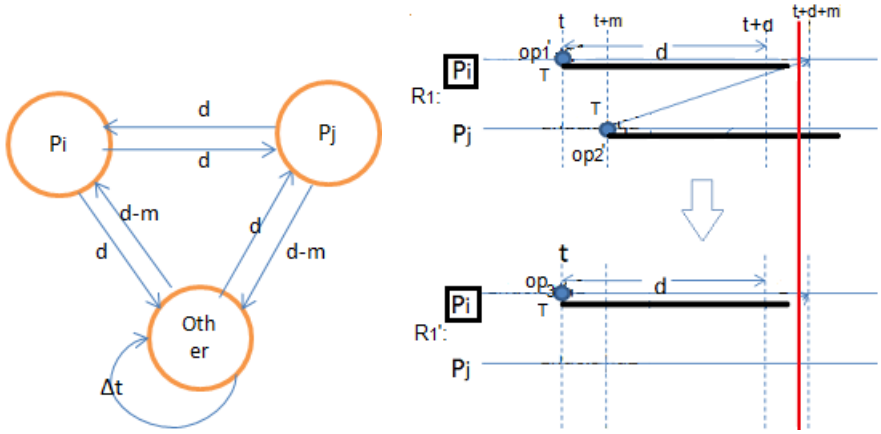
are also the same as in R_1; p_i executes an operation $op_3 \in OP$ with argument arg_1 invoked at real time t (local time T). No other operations are invoked.

Then runs R_1 and R_1' are both admissible because:

- The maximum local clock skew is $m = \min\{\epsilon, u, d/3\}$, which is not larger than ϵ.
- Because $m = \min\{\epsilon, u, d/3\}$ is not larger than u, letting message delay be $d-m$ or d is admissible.

Step 1.1: We will show $op_1' = op_1$ (① and ② in Fig. 6), i.e., that the value returned in R_1 for op_1' is ret_1.

In run R_1', by the definition of a deterministic object, op_3 has to return ret_1



(a) Message Delays in R_1 and R_1'

(b) R_1 and R_1'

Fig. 7. Step 1 in Theorem C.1

because there is no other concurrent operations. Therefore $op_3 = op_1$.

In run R_1, there are two concurrent operations op_1' and op_2'. Since $m \leq d/3$, it follows that $D_{j,i} \geq d$, meaning that p_i cannot "learn about" op_2' until real time $t + d + m$. Then before $t + d + m$, the view of p_i is the same in R_1 and in R_1'. Therefore, op_1' returns the same value ret_1 as op_3 in R_1', and so $op_1' = op_3 = op_1$. (Fig. 7(b))

Step 1.2: We will show $op_2' \neq op_2$(③ in Fig. 6).

By the assumed correctness of the algorithm, at least one of the below must be legal: $\rho \circ op_1' \circ op_2'$ and $\rho \circ op_2' \circ op_1'$. According to the definition of strongly immediately non-self-commuting operation, $\rho \circ op_1 \circ op_2$ and $\rho \circ op_2 \circ op_1$ are both illegal. We already proved $op_1' = op_1$. If $op_2' = op_2$, it would mean neither $\rho \circ op_1' \circ op_2'$ nor $\rho \circ op_2' \circ op_1'$ is legal, contradicting that at least one of $\rho \circ op_1' \circ op_2'$ and $\rho \circ op_2' \circ op_1'$ must be legal.

Step 1.3: We will show that $\rho \circ op_2' \circ op_1'$ is illegal and thus $\rho \circ op_1' \circ op_2'$ must be legal.

Because $op_2' \neq op_2$ and $\rho \circ op_2$ is legal, based on the definition of deterministic object, $\rho \circ op_2'$ must be illegal, which means $\rho \circ op_2' \circ op_1'$ is illegal.

Step 2: Define run $R_2' = shift(R_1, \vec{x})$, where $x_j = -m$ and $x_k = 0$ for all $k \neq j$, which means p_i executes op_1' with argument arg_1 invoked at real time t (local time T), and p_j executes op_2' with argument arg_2 invoked at real time t (local time T).

Now we apply Lemma B.1 here. According to formula (1), $d'_{j,i}$ in R_2' is $d + m$, and it is the only invalid message delay. Because op_1' and op_2' are both invoked at real time t in R_2', the earliest possible message from p_j to p_i is sent no earlier than t. By setting δ to $d - m$, we get $t^* = t + \min\{d + m, d - m\} = t + d - m$. Let $R_2'' = chop(R_2', d - m)$. By Lemma B.1, R_2'' is admissible. (In Fig. 8, the cross signs are where each view is chopped.)

We can extend R_2'' to a complete admissible run R_2 by letting messages from p_j to p_k cost $\delta = d - m$ time. The invocations of op_1' and op_2' happen in R_2'', but the responses of them may not. So in R_2, p_i invokes an operation $op_1'' \in OP$ with arg_1 at real time t, and p_j invokes an operation $op_2'' \in OP$ with arg_2 at real time t, but the response of op_1'' (respectively op_2'') may be different from that of op_1' (respectively op_2').

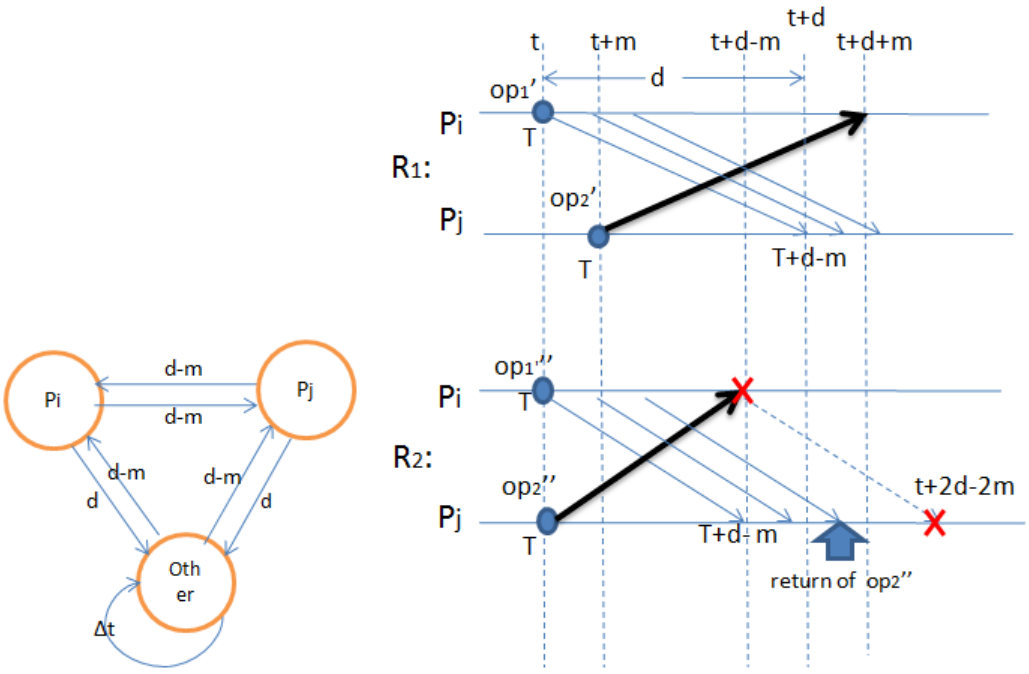
However, we will now prove that $op_1'' = op_1' = op_1$ and $op_2'' = op_2'$.

Step 2.1: First we show $op_2'' = op_2'$ (④ in Fig. 6).

For process p_j, $D'_{i,j} = d_{i,j} = d - m$, because $d - m$ is the smallest message delay in R_2''. Then $t^* + D'_{i,j} = t + d - m + d - m \geq t + d + m$, and $V_j \in R_2''$ ends after $t + d + m$. Because op_2'' returns before $t + d + m$, op_2'' responds within R_2'', which is a prefix of $shift(R_1, \vec{x})$. Therefore, op_2'' returns the same value as op_2' and $op_2'' = op_2'$.

Step 2.2: Now we show $op_1'' = op_1$(⑤ in Fig. 6).

Since $op_2'' \neq op_2$ and $\rho \circ op_2$ is legal, by the definition of a deterministic object, $\rho \circ op_2''$ is illegal. So $\rho \circ op_2'' \circ op_1''$ is illegal, and then by the assumed correctness of the



(a) Message Delays in R_2 (b) R_1 and R_2

Fig. 8. Step 2 in Theorem C.1

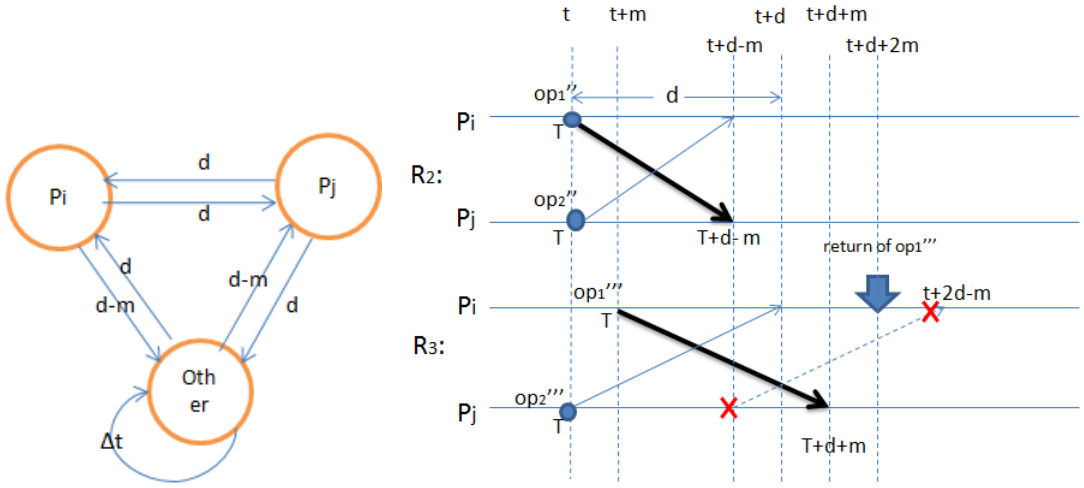
algorithm, $\rho \circ op_1'' \circ op_2''$ must be legal. Considering $\rho \circ op_1$ is legal and op_1'' has the same argument as op_1, $op_1'' = op_1$, otherwise it contradicts the definition of deterministic object.

Step 3: Let run $R_3' = shift(R_2, \vec{x})$, where $x_i = m$ and $x_k = 0$ for all $k \neq i$, which means p_i executes op_1'' with argument arg_1 invoked at real time $t + m$ (local time T), and p_j executes op_2'' with argument arg_2 invoked at real time t (local time T).

Now we apply Lemma B.1 here. According to formula (1), $d''_{i,j}$ is $d - 2m$, which may violate the message delay constraint. The earliest possible message from p_i to p_j is sent no earlier than real time $t + m$, because op_1'' is invoked at real time $t + m$ in R_3', and op_2'' is invoked at real time t, meaning the earliest message to p_i is received no earlier than $t + d - m$, which is larger than $t + m$ because $m \leq d/3$. Then still

by setting $\delta = d - m$, we get $t^* = t + m + \min\{d - 2m, d - m\} = t + d - m$. Let $R_3'' = chop(R_3', d - m)$. By Lemma B.1, R_3'' is admissible. (In Fig. 9, the cross signs are where each view is chopped.)

We can extend R_3'' to a complete admissible run R_3 by letting messages sent from p_i to p_j cost $d > \delta$ time. Similarly to Step 2, the invocations of op_1'' and op_2'' happen in R_2'', but the responses of them may not. So p_i invokes an operation $op_1''' \in OP$ with arg_1 at real time $t + m$, and p_j invokes an operation $op_2''' \in OP$ with arg_2 at real time t, but the response of op_1''' (respectively op_2''') may be different from that of op_1'' (respectively op_2'').



(a) Message Delays in R_3 and R_3'

(b) R_2 and R_3

Fig. 9. Step 3 in Theorem C.1

Similarly to step 2.2, we can prove that $op_1''' = op_1'' = op_1$(⑥ in Fig. 6). Then similarly to the last two paragraphs in Step 1, we can show that $op_2''' \neq op_2$(⑦ in Fig. 6) and $\rho \circ op_1''' \circ op_2'''$ is legal.

Step 4: We define a run R_3''' as follows. Let all processes in run R_3''' have the

same clock functions and message delays as in R_3. Then R_3''' is also admissible.

In run R_3''': p_j executes op_4 with argument arg_2 invoked at real time t. No other operations are invoked.

In R_3''', because there is only one operation op_4, by the definition of a deterministic object, op_4 must return ret_2. So $op_4 = op_2$(⑨ in Fig. 6). Next we prove $op_2''' = op_4 = op_2$(⑧ in Fig. 6).

Similar to Step 1, because it is impossible for p_j to get any information about op_1''' before op_2''' returns, p_j has the same local view in R_3 and R_3''' until op_2''' returns. Therefore, op_2''' in R_3 returns what op_4 returns and $op_2''' = op_4 = op_2$. This contradicts the conclusion in Step 3 that $op_2''' \neq op_2$. □

D. Eventually Non-self-last-permuting Operation Types

By studying the commutativity property, the time lower bounds for eventually non-self-commuting operation types such as write, push, enqueue have been proved to be $u/2$ [3]. Considering the scale of the ever growing shared object system, we extend commutativity to permutations for more than two concurrent operations. In the following theorem, we prove using the standard shift technique, that for a group of characterized operation types, which also includes write, push and enqueue, the lower bound of operations in this group is $(1-1/k)u$ ($k \leq n$ is a positive integer determined by the property of the operations). More specifically, for those immediately non-self-last-permuting operations such as write on a register, push on a stack and enqueue on a queue, $k = n$ and the lower bounds for them are $(1 - 1/n)u$.

Theorem D.1 *Let OP be an immediately self-commuting operation type, for which there exist an integer $k \geq 2$, an operation sequence ρ, and k distinct operations $op_i(arg_i, ret_i) \in OP$, $0 \leq i \leq k-1$, such that for all $0 \leq i \leq k-1$, $\rho \circ op_i$ is*

legal, and for any two legal permutations π *and* π' *of* $\{op_i : 0 \leq i \leq k-1\}$ *with* $last(\pi) \neq last(\pi')$, π *and* π' *are not equivalent. Then the time complexity of* OP *in any implementation of* OP *in a system with* $n \geq k$ *processes,* ϵ*-bounded clocks, and message delays in* $[d-u, d]$ *is at least* $(1-1/k)u$.

PROOF. Suppose in contradiction there is an implementation of such an operation type OP in such a system with time complexity less than $(1-1/k)u$. Since $\rho \circ op_i$ is legal for each i, and OP is immediately self-commuting, a simple inductive proof shows that for any permutation π of the k operations, $\rho \circ \pi$ is legal.

Step 1: We construct run R_1 with the process set Π in which:

- all the processes have the same clock function;
- For all $0 \leq i, j \leq k-1$, $i \neq j$, $d_{i,j} = d - [(i-j)\%k/k] \cdot u$. For all $k \leq l \leq n-1$, all $0 \leq l' \leq n-1$ and $l \neq l'$, $d_{l,l'} = d_{l',l} = d - u/2$ (shown in Fig. 10).

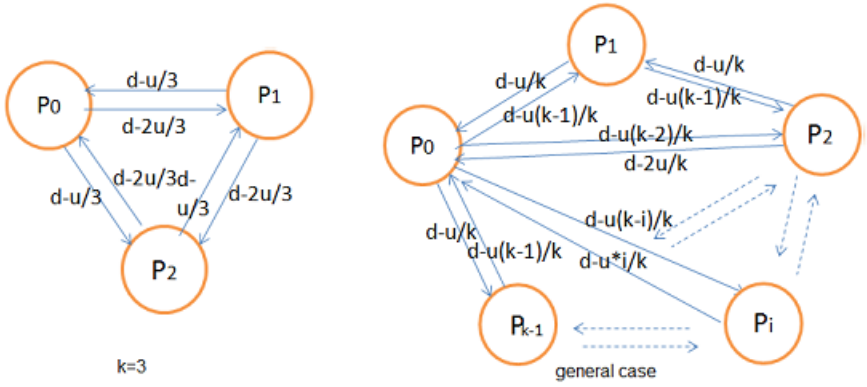


Fig. 10. Message Delays in $R_1(R_1')$

In run R_1(Fig. 11), each process p_i $(0 \leq i \leq k-1)$ executes op_i at real time t (local clock T). No other operations are invoked. Then each p_i ends before $t+u$.

R_1 is admissible because:

1. The maximum local clock skew is 0, smaller than ϵ.

2. Because $d - u < d - j \cdot u/k \leq d$, all the message delays are admissible.

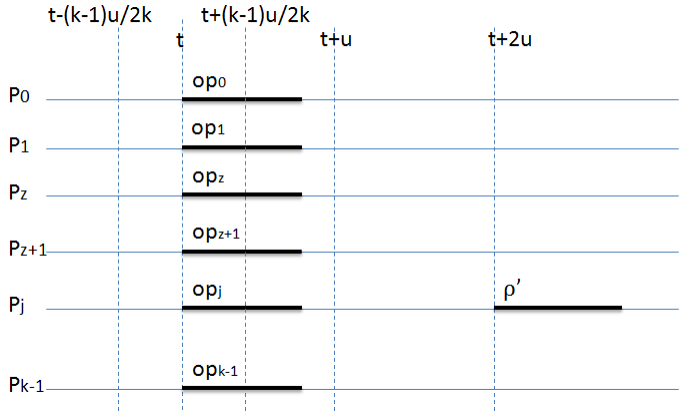


Fig. 11. Run $R_1(R_1')$

Step 1.1: By the definition of the deterministic object and the assumed correctness of the algorithm, there exists at least one legal permutation π for the k operations in R_1, such that for any operation sequence ρ^*, if $\rho \circ \pi \circ \rho^*$ is legal, then when the operations in ρ^* are invoked sequentially after $t + 2u$ (after all the operations in R_1 return), each operation returns the same value as in ρ^*.

Step 1.2: Without loss of generality, assume $last(\pi) = op_z$.

Step 2: Let run $R_2 = shift(R_1, \vec{x})$ where $x_i = [-(k-1)/2 + (z-i)\%k/k] \cdot u$ $(0 \leq i \leq k-1)$. According to formula (1), $d'_{i,j} = d - [(i-j)\%k/k] \cdot u - [-(k-1)/2 + (z-i)\%k/k] \cdot u + [-(k-1)/2 + (z-j)\%k/k] \cdot u = d - [(i-j)\%k/k] \cdot u - [(z-i)\%k/k] \cdot u + [(z-j)\%k/k] \cdot u$ (shown in Fig. 13 and Fig. 14).

- If $i < j \leq z$, $d'_{i,j} = d - [(i-j+k)/k] \cdot u - [(z-i)/k] \cdot u + [(z-j)/k] \cdot u = d - u$.

- If $i \leq z < j$, $d'_{i,j} = d - [(i-j+k)/k] \cdot u - [(z-i)/k] \cdot u + [(z-j+k)/k] \cdot u = d$.
- If $z < i < j$, $d'_{i,j} = d - [(i-j+k)/k] \cdot u - [(z-i+k)/k] \cdot u + [(z-j+k)/k] \cdot u = d - u$.
- If $j < i \leq z$, $d'_{i,j} = d - [(i-j)/k] \cdot u - [(z-i)/k] \cdot u + [(z-j)/k] \cdot u = d$.
- If $j \leq z < i$, $d'_{i,j} = d - [(i-j)/k] \cdot u - [(z-i+k)/k] \cdot u + [(z-j)/k] \cdot u = d - u$.
- If $z < j < i$, $d'_{i,j} = d - [(i-j)/k] \cdot u - [(z-i+k)/k) \cdot u + [(z-j+k)/k] \cdot u = d$.

(A simple example of this shift is in Fig. 12.)

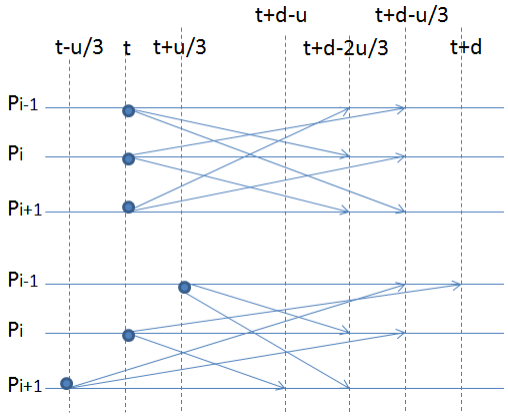


Fig. 12. Example of Standard Time Shift of R_1

R_2 is admissible because:

1. The maximum local clock skew is $[-(k-1)/2 + (k-1)]/k \cdot u - [-(k-1)/2k]u = (1-1/k)u$, no larger than ϵ, because the optimal ϵ is $(1-1/n)u \geq [1-(k-1)/k]u$.
2. All the message delays are admissible:
   - For $0 \leq i, j \leq k-1$, $d'_{i,j}$ is either d or $d-u$, which are both admissible.

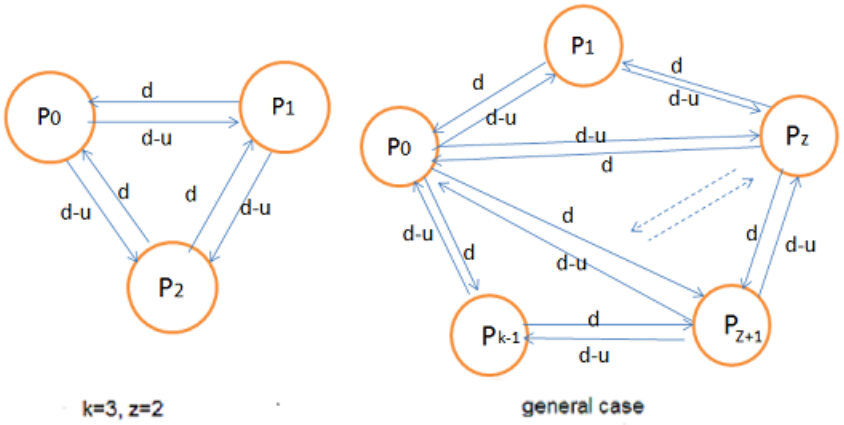


Fig. 13. Message Delays in $R_2(R_2')$

- For $k \leq l \leq n-1, 0 \leq l' \leq n-1$, $d_{l,l'} = d - u/2 - x_l + x_l' = d - u/2 + x_l'$ (because $x_l = 0$). p_l' is shifted by x_l' where $|x_l'| < u/2$, so $d - u \leq d_{l,l'} \leq d$. In the same way, we can prove $d - u \leq d_{l',l} \leq d$.

Step 2.1: By the definition of the deterministic object and the assumed correctness of the algorithm, there exists at least one legal permutation π' for the k operations in R_2, such that for any operation sequence ρ^*, if $\rho \circ \pi' \circ \rho^*$ is legal, then when the operations in ρ^* are invoked sequentially after $t+u$ (after all the operations in R_2 return), each operation returns the same value as in ρ^*.

Step 2.2: We show op_z cannot be the last operation in π'. Because p_z is shifted by $-[(k-1)/2k] \cdot u$ and $p_{(z+1)\%k}$ is shifted by $[(k-1)/2k] \cdot u$, $op_{(z+1)\%k}$ is invoked after op_z returns. $([(k-1)/2k - (-(k-1)/2k)] \cdot u = [(k-1)/k] \cdot u > |OP|)$. So in any permutation π' that respects the real time sequence in R_2, op_z cannot be the last operation.

Step 3: From the two steps above we have $last(\pi) \neq last(\pi')$. According to the assumptions in the theorem, because $last(\pi) \neq last(\pi')$, there exists an operation

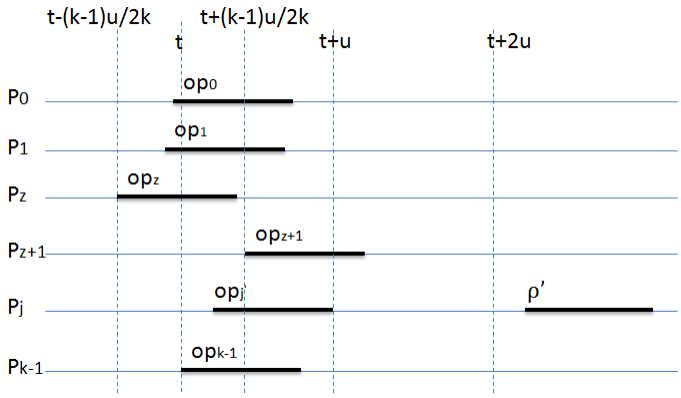


Fig. 14. Run $R_2(R_2')$

sequence ρ' such that one of the below is legal while the other is illegal:

(1) $\rho \circ \pi \circ \rho'$

(2) $\rho \circ \pi' \circ \rho'$

Without loss of generality, assume $\rho \circ \pi \circ \rho'$ is legal and thus $\rho \circ \pi' \circ \rho'$ is illegal. Let R_1 be extended to R_1' where a process invokes operations in ρ' sequentially after $t + 2u$. And let $R_2' = shift(R_1', \vec{x})$. Then it is easy to prove that R_2' is an extension of R_2, because the shift amounts from R_1 to R_2 and from R_1' to R_2' are the same. R_2' is admissible but $\rho \circ \pi' \circ \rho'$ is illegal, contradicting the assumed correctness of the algorithm.

Step 4: By repeating Step 3 for every π' which satisfies the condition in Step 2.1, we can prove there is a contradiction for every π', which means π' does not exist, and so op_z cannot be the last operation in π. By repeating Step 2 - Step 3 for every π which satisfies the condition in Step 1.1, we can prove none of the k operations can be the last one in π, which means π does not exist, contradicting the assumed correctness of the algorithm.

□

This lower bound can be applied to the message passing system with n processes where n is a finite integer. If $k = 2$, then $(1 - 1/k)u = u/2$, matching the time lower bound proof for non-self-commuting property, also matching that non-self-commuting is one special case of non-self-permuting. And for the write, enqueue and push operation, we know that $k = n$. So the lower bound for them is $(1 - 1/n)u$, matching the optimal local clock skew.

E. Immediately Non-commuting Pairs of Operation Types

For any two operation types OP_1 and OP_2 which immediately do not commute, $|OP_1| + |OP_2| \geq d$ [3]. In the perfect synchronous system, it has been proved to be a tight time bound for immediately non-commuting pure mutators and pure accessors [3]. In the partially synchronous system, the lower bound still holds.

Compared with the eventually self-commuting operation types, we are more interested in eventually non-self-commuting operation types, because usually an object has several operation types and not all pairs of two pure mutators are eventually commuting. Suppose two operation types OP_1 and OP_2 eventually do not commute with each other and OP is an eventually non-self-commuting operation type. Then having two concurrent operations $op_1 \in OP_1$ and $op_2 \in OP_2$ is similar to the case of having two concurrent operations $op_1, op_2 \in OP$. By solving the linearizability problem for the eventually non-self-commuting operations, the problem for a pair of eventually non-commuting operations can also be solved.

Mavronicolas and Roth [7] prove for write and read on a shared object that the total time lower bound is at least $d + \min\{\epsilon, u\}/2$. However, this result only applies for certain class of algorithms with the constraints – object symmetric, which means each process handles activity involving a certain object in precisely the same way it

handles activity on any other. We want to get rid of that constraint in our work.

Below we prove $|OP_1| + |OP_2| \geq d + \min\{\epsilon, u, d/3\}$ where OP_1 is immediately self-commuting but eventually non-self-commuting and non-overwriting, and OP_2 is a pure mutator which immediately does not commute with OP_1. The theorem can be applied to push and peek on a stack, and enqueue and peek on a queue. However, it cannot be applied to the write and read operations on a register.

We let OP_1 be immediately self-commuting, because if OP_1 is immediately non-self commuting, the lower bound has been proved to be $d+\min\{\epsilon, u, d/3\}$(Theorem C.1), which is the same as the lower bound we prove below.

We let OP_2 be a pure accessor (recall that a pure accessor does not modify the object) because:

1. If OP_2 modifies the object, it can be an overwriter or it can reverse the modifications of OP_1. Then OP_2 has the same effect as an overwriter. Although we can add constraints on OP_2 to exclude these cases, the theorem itself will be too complicated.

2. For the commonly used pair of immediately non-commuting operations such as push and pop, enqueue and dequeue, we have already proved that the lower bound for pop and dequeue is $d + \min\{\epsilon, u, d/3\}$(Theorem C.1), which is the same as in the following proof.

Under these assumptions of OP_1 and OP_2, we use two operations $op_1 \in OP_1$ and $op_1' \in OP_1$. There are three ways to see if op_1 should be put before op_1' in the legal permutation of concurrent operations:

1. op_1 completes before op_1' is invoked.

2. An operation $op_2 \in OP_2$ invoked after op_1 and op_1' complete can tell which one should be arranged as the first operation.

3. An operation $op_3 \in OP_2$ that reflects the modification of op_1 has been completed on the object while the modifications of op_1' are not executed.

We can construct counter-examples to prove the violation of linearizability by using the three points above. For example, we can let op_1' be invoked after op_1 completes, then let another operation op_2 be invoked after op_1' completes. If the return value of op_2 reflects that op_1' should be arranged before op_1, then there is a contradiction.

However, the third point has a problem. Suppose OP_1 is the write operation on a read/write register and OP_2 is the read operation on the read/write register. If the register is initialized with zero, and $write(1)$, $write(2)$ and $read(2)$ are concurrent operations, then there are two legal permutations - $write(2) \circ read(2) \circ write(1)$ and $write(1) \circ write(2) \circ read(2)$. So if OP_1 can overwrite the whole state of the object, even if op_3 reflects that op_1 has been completed, it cannot tell whether op_1' has not been executed or op_1' has been executed but overwritten by op_1. That is why the non-overwriting property may impact the time lower bound.

Theorem E.1 *For immediately self-commuting operation type OP and pure accessor AOP, if there exist an operation sequence ρ, operation $op_1, op_2 \in OP$ and $aop_1, aop_2, aop_3 \in AOP$ such that:*

A: one of the below is legal while the other one is illegal:

- $\rho \circ op_1 \circ aop_1$
- $\rho \circ op_2 \circ op_1 \circ aop_1$

B: one of the below is legal while the other one is illegal:

- $\rho \circ op_2 \circ aop_2$
- $\rho \circ op_1 \circ op_2 \circ aop_2$

and C: one of the below is legal while the other one is illegal:

- $\rho \circ op_1 \circ op_2 \circ aop_3$
- $\rho \circ op_2 \circ op_1 \circ aop_3$

then $|OP| + |AOP| \geq d + \min\{\epsilon, u, d/3\}$ *in an n-process*($n \geq 3$) *message passing system with message delay bound* $[d-u, d]$ *and local clock skew bound* ϵ.

PROOF. Assume in contradiction that there are two operation types OP and AOP which satisfy the properties in the theorem but $|OP|+|AOP| < d+m$ time where $m = \min\{\epsilon, u, d/3\}$. According to the theorem statement, we assume $op_1 = OP(arg_1, ret_1)$, $op_2 = OP(arg_2, ret_2)$ and $aop_1 = AOP(aarg_1, aret_1)$, $aop_2 = AOP(aarg_2, aret_2)$, $aop_3 = AOP(aarg_3, aret_3)$.

Because either $\rho \circ op_1 \circ op_2 \circ aop_3$ or $\rho \circ op_2 \circ op_1 \circ aop_3$ is legal, at least one of $\rho \circ op_1 \circ op_2$ and $\rho \circ op_2 \circ op_1$ is legal. If $\rho \circ op_1 \circ op_2$ is legal, $\rho \circ op_2 \circ op_1$ is also legal, because op_1 and op_2 are immediately self-commuting operations. Vice versa. So we have both of the below are legal:

- $\rho \circ op_1 \circ op_2$
- $\rho \circ op_2 \circ op_1$

The counter-example is shown as below. (Similarly to theorem C.1, we let one process invokes ρ in sequence and wait until the system becomes quiescent for a long enough time. And we ignore ρ in the following runs.) Proof structure refers to Fig. 15.

Step 1: We construct run R_1 as follows (Fig. 16). Let p_i, p_j and p_k be three specific processes and p_l denotes any other process:

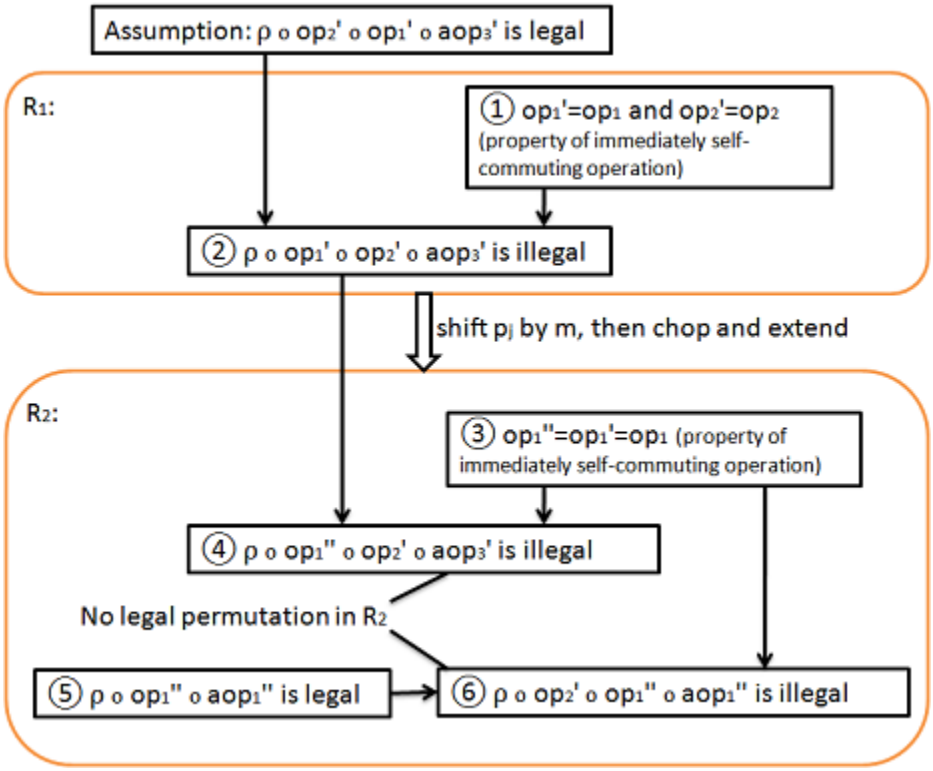


Fig. 15. Proof Structure of Theorem E.1

- All the processes have the same local clock function.
- $d_{i,k} = d_{i,l} = d_{j,k} = d_{j,l} = d$, and $d_{i,j} = d_{j,i} = d_{k,i} = d_{l,i} = d_{k,j} = d_{l,j} = d - m$. $d_{k,l}$ and $d_{l,k}$ is any value in the range $[d - u, d]$.

R_1 is admissible because:

1. the maximum local clock skew is $0 < \epsilon$; and
2. letting the message delay be $d - m$ or d is admissible.

In run R_1, p_i executes $op'_1 \in OP$ with arg_1 invoked at time t; p_j executes $op'_2 \in OP$ with arg_2 invoked at time t. Suppose op'_1 returns at real time t_1 and op'_2 returns at real time t_2. Let $t_{max} = max\{t_1, t_2\}$. Then p_i executes $aop'_1 \in AOP$ with $aarg_1$

invoked at real time t_{max}. p_j executes $aop'_2 \in AOP$ with $aarg_2$ invoked at t_{max}. Process p_k executes $aop'_3 \in AOP$ with $aarg_3$ invoked at real time $t_{max} + m$. No other operations are invoked.

Since aop'_1, aop'_2 and aop'_3 are invoked after op'_1 and op'_2 return, in any legal permutation of the five operations, aop'_1, aop'_2 and aop'_3 must be put after op'_1 and op'_2. Because $|OP| + |AOP| < d + m$, aop'_1 and aop'_2 return before $t + d + m$ and aop'_3 returns before $t + d + 2m$. Since aop'_1 and aop'_2 are pure mutators, it doesn't matter if they are before or after aop'_3 in the linearized permutation. Therefore, by the assumed correctness of the algorithm, at least one of $\rho \circ op'_1 \circ op'_2 \circ aop'_3$ and $\rho \circ op'_2 \circ op'_1 \circ aop'_3$ must be legal. Without loss of generality we assume $\rho \circ op'_2 \circ op'_1 \circ aop'_3$ is legal.

Step 1.1: First we show $op'_1 = op_1$ and $op'_2 = op_2$.

Because op'_1 and op'_2 are before aop'_1, aop'_2 and aop'_3 in any legal permutation, the assumed correctness of the algorithm ensures at least one of the below is legal:

- $\rho \circ op'_1 \circ op'_2$
- $\rho \circ op'_2 \circ op'_1$

If $\rho \circ op'_1 \circ op'_2$ is legal, by the definition of the deterministic object, because $\rho \circ op_1 \circ op_2$ is legal, $op'_1 = op_1$. Then because of the same reason, $op'_2 = op_2$. If $\rho \circ op'_2 \circ op'_1$ is legal, we can get the same conclusion in the similar way.

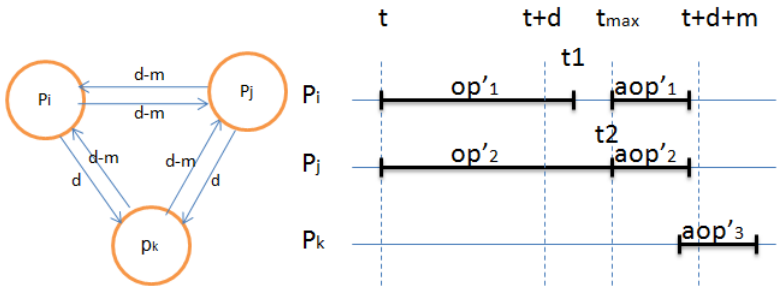
Step 1.2: Now we prove $\rho \circ op'_1 \circ op'_2 \circ aop'_3$ must be illegal by using assumption C in the theorem with the case analysis below(② in Fig. 15).

Case 1: $\rho \circ op_2 \circ op_1 \circ aop_3$ is legal and $\rho \circ op_1 \circ op_2 \circ aop_3$ is illegal.

Because $op'_1 = op_1$, $op'_2 = op_2$, and $\rho \circ op'_2 \circ op'_1 \circ aop'_3$ is legal, by the definition of the deterministic object, we get $aop'_3 = aop_3$. Therefore $\rho \circ op'_1 \circ op'_2 \circ aop'_3$ is illegal because $\rho \circ op_1 \circ op_2 \circ aop_3$ is illegal.

Case 2: $\rho \circ op_2 \circ op_1 \circ aop_3$ is illegal and $\rho \circ op_1 \circ op_2 \circ aop_3$ is legal.

Because $op'_1 = op_1$, $op'_2 = op_2$, and $\rho \circ op'_2 \circ op'_1 \circ aop'_3$ is legal, by the definition of the deterministic object, we get $aop'_3 \neq aop_3$. Therefore $\rho \circ op'_1 \circ op'_2 \circ aop'_3$ is illegal because $\rho \circ op_1 \circ op_2 \circ aop_3$ is legal.



(a) Message Delays in R_1 (b) R_1

Fig. 16. Step 1 in Theorem E.1

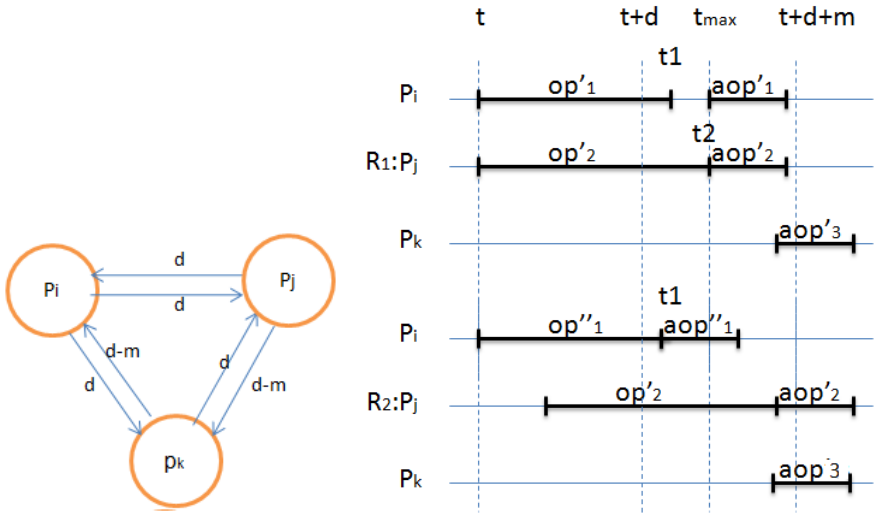
Step 2: Define run $R'_2 = shift(R_1, \vec{x})$, where $x_j = m$ and $x_l = 0$ for all $l \neq j$, which means p_i executes op'_1 with arg_1 invoked at real time t, p_j execute op'_2 with arg_2 invoked at time $t + m$, aop'_1 and aop'_3 are still executed at the same real time as in R_1, and aop'_2 is executes m real time later than in R_1 (refer to Fig. 17).

Now we apply Lemma B.1 here. According to formula (1), $d'_{j,i}$ is $d - 2m$ which may violate the message delay constraint, and it is the only possible invalid message delay. The earliest possible message from p_j to p_i is sent no earlier than $t+m$, because (1) op'_2 is invoked at $t + m$ in R'_2; (2) the earliest message in R'_2 is sent after t, and so if it is received by p_j, it is received after $t+d-m > t+m$. Let $R''_2 = chop(R'_2, d-m)$. In the chopping procedure, $t^* = t + m + \min\{d - 2m, d - m\} = t + d - m$. By Lemma B.1, R''_2 is admissible.

$V_i \in R''_2$ ends just before $t + d - m$. So the invocation of op'_1 occurs in R''_2 but the response may not. For process p_j, because $d'_{i,j} = (d - m) - 0 + m = d$ and

$d'_{i,k} = d'_{i,l} = d$, we get $D'_{i,j} = d$. So $V_j \in R''_2$ ends just before $t+d-m+d \geq t+d+2m$ (because $m \leq d/3$). Because aop'_2 returns before $t+d+m$ in R_1, we get aop'_2 returns before $t+d+2m$ in R'_2. So V_j ends after aop'_2 returns. Similarly, we can prove $V_k \in R''_2$ ends after aop'_3 returns.

We can extend R''_2 to a complete admissible run R_2 by setting the delay for messages from p_j to p_i to be $d > \delta$. The response of op'_1 may not occur in V_i and it may be different from ret'_1. So we call the operation invoked by p_i at real time t op''_1. In R_2, if the invocation of aop'_1 is not in R''_2 (meaning $t+d-m < t_{max}$), let p_i invoke operation $aop''_1 \in AOP$ with $aarg_1$ after op''_1 returns and V_i ends. Depending on whether op''_1 returns later or V_i ends later, aop''_1 is invoked immediately after the later one of them occurs. So aop''_1 is invoked at $\max\{\min\{t_{max}, t+d-m\}, t+|op''_1|\}$. Here $t+d-m$ is the ending time of V_i, and $t+|op''_1|$ is the response time of operation op''_1.



(a) Message Delays in R_2

(b) R_1 and R_2

Fig. 17. Step 2 in Theorem E.1

Step 2.1: First we show $op_1'' = op_1$ (③ in Fig. 15) and thus $\rho \circ op_1'' \circ op_2' \circ aop_3'$ is illegal (④ in Fig. 15).

By the assumed correctness of the algorithm, at least one of $\rho \circ op_1'' \circ op_2$ and $\rho \circ op_2 \circ op_1''$ is legal. No matter which one is legal, since we have that $\rho \circ op_1 \circ op_2$ and $\rho \circ op_2 \circ op_1$ are both legal, by the definition of the deterministic object, it follows that $op_1'' = op_1$. And since in Step 1.2 we have proved $\rho \circ op_1' \circ op_2' \circ aop_3'$ is illegal, $\rho \circ op_1'' \circ op_2' \circ aop_3'$ is illegal.

Step 2.2: We show $\rho \circ op_2' \circ op_1'' \circ aop_1''$ is illegal(⑥ in Fig. 15).

Step 2.2a: First we prove aop_1'' returns before $t + d + m$. The invocation time of aop_1'' is $\max\{\min\{t_{max}, t + d - m\}, t + |op_1''|\}$.

1. Case 1: $t_{max} \leq t + d - m$. Then aop_1'' is invoked before $t + d - m$ and op_1'' responds before $t + d - m$, meaning $t + |op_1''| < t + d - m$. Therefore, aop_1'' is invoked at t_{max}.

   (a) Case 1.1: $t_{max} = t_1$. Then aop_1'' is invoked immediately after op_1'' responds. Because $|op_1''| + |aop_1''| < d + m$ and op_1'' is invoked at real time t, aop_1'' returns before $t + d + m$.

   (b) Case 1.2: $t_{max} = t_2$. Then $|op_2'| = t_{max} - t$. If aop_1'' does not return before $t + d + m$, then $|op_2'| + |aop_1''| \geq (t_{max} - t) + (t + d + m - t_{max}) = d + m$, contradicting that $|OP| + |AOP| < d + m$.

2. Case 2: $t + d - m < t_{max}$ and $t + d - m \leq t + |op_1''|$. Then aop_1'' is invoked at $t + |op_1''|$, immediately after op_1'' responds. Similarly to case 1.1, aop_1'' returns before $t + d + m$.

3. Case 3: $t + d - m < t_{max}$ and $t + d - m > t + |op_1''|$. Then aop_1'' is invoked at $t + d - m$.

(a) Case 3.1: $t_{max} = t_1$. Then op'_1 returns at $t_{max} > t + d - m$. So op''_1 cannot return before $t+d-m$, because until $t+d-m$, p_i has the same local view in R_1 and R_2. This contradicts $t + d - m > t + |op''_1|$.

(b) Case 3.2: $t_{max} = t_2$. Similarly to case 1.2, if aop''_1 does not return before $t+d+m$, we can get $|op'_2|+|aop''_1| \geq (t_{max}-t)+(t+d+m-(t+d-m)) = t_{max} - t + 2m > (t + d - m) - t + 2m > d + m$, contradicting that $|OP| + |AOP| < d + m$.

Step 2.2b: Now we analyze R_2 from the view of p_i. Because op''_2 is invoked at $t + m$ and $D'_{j,i} = d$, it is impossible for p_i to get any information about op''_2 before $t + d + m$, which is after aop''_1 returns. So without learning the existence of op''_2, p_i returns aop''_1 such that $\rho \circ op''_1 \circ aop''_1$ is legal. Because $op''_1 = op_1$, $op'_2 = op_2$ and $\rho \circ op''_1 \circ aop''_1$ is legal, similarly to the analysis in step 1.2, by using assumption A in the theorem, $\rho \circ op'_2 \circ op''_1 \circ aop''_1$ is illegal.

Contradiction: From step 2.1, because $\rho \circ op''_1 \circ op'_2 \circ aop'_3$ is illegal and aop'_3 is invoked after op'_2 returns, by the assumed correctness of the algorithm, in any legal permutation of the operations in R_2, op''_1 must be put after op'_2. From step 2.2, because $\rho \circ op'_2 \circ op''_1 \circ aop''_1$ is illegal and aop''_1 is invoked after op''_1 returns, by the assumed correctness of the algorithm, in any legal permutation of the operations in R_2, op'_2 is after op''_1, contradicting the conclusion from step 2.1 that op''_1 must be put after op'_2. Therefore, R_2 is an admissible run but there is no legal permutation for the operations in R_2, contradicting the assumed correctness of the algorithm. □

Remark:

1. In theorem E.1, although we never explicitly say OP and AOP immediately do not commute with each other, they are immediately non-commuting operations. The explanation is as below.

Suppose in contradiction, OP and AOP are immediately commuting operations. From the assumption in theorem E.1, there exist instances op_1, op_2 of OP, and instance aop_3 of AOP, such that one of the below is legal while the other one is illegal:

(1) $\rho \circ op_1 \circ op_2 \circ aop_3$

(2) $\rho \circ op_2 \circ op_1 \circ aop_3$

Without loss of generality, let's assume $\rho \circ op_1 \circ op_2 \circ aop_3$ is legal. Since OP and AOP are immediately commuting, we can switch op_2 and aop_3 and get $\rho \circ op_1 \circ aop_3 \circ op_2$ is legal. In a similar way, we can continue to switch op_1 and aop_3, without affecting op_2 because aop_3 is a pure accessor, and we get $\rho \circ aop_3 \circ op_1 \circ op_2$ is legal. Next, because op_1 and op_2 are immediately commuting, we exchange the position of them and get $\rho \circ aop_3 \circ op_2 \circ op_1$ is legal. Finally, we switch aop_3 back to be the last operation and get $\rho \circ op_2 \circ op_1 \circ aop_3$ is legal, contradicting the assumption in the theorem. Therefore, OP and AOP are immediately non-commuting operations.

2. In theorem E.1, we assumed $\rho \circ op'_2 \circ op'_1 \circ aop'_3$ is legal in Step 1, and then use assumptions A and C to get the contradiction in the counter-example. If $\rho \circ op'_1 \circ op'_2 \circ aop'_3$ is legal, then p_i is shifted by m and we will use assumptions B and C to get the contradiction.

CHAPTER V

UPPER BOUNDS FOR LINEARIZABLE OBJECTS

In this chapter, we will present an implementation for an arbitrary data type with all types of operations. We group all the operations on a shared object into three types: pure accessors AOP, pure mutators MOP and all the other operations OOP.

By a result in [6], the optimal value of ϵ is $(1 - 1/n)u$, which is smaller than u. And there already exist some implementations for optimal ϵ. The implementation for a shared object below is built on a message passing system whose clocks are synchronized to within the optimal ϵ and have no drift.

A. Main Idea of the Implementation

Each process keeps a copy of the object and we assume each copy is initialized with the same initial value of the object. The events in each process are triggered by operation invocation, message reception and timer expiration.

We start a timer with the function $set\_timer(counter, \langle op, arg, ts\rangle, action)$. Each timer is set for an operation, so we add the information of the operation into the timer in the format $\langle op, arg, ts\rangle$, where arg is the argument of op, and ts is the timestamp of op. Each timestamp is in the format $\langle clock\_time, process\_id\rangle$ where $clock\_time$ is the local clock time and $process\_id$ is the id of op's invoking process. The timer counts down from the $counter$ value to 0. When it reaches 0, the timer expires and $expire\_timer(\langle op, arg, ts\rangle, action)$ occurs. According to op and $action$, the timer expiration will trigger some events. Each timer can be canceled with the function $cancel\_timer(\langle op, arg, ts\rangle, action)$.

The local clock time of process p_i is indicated by $local\_time_i$.

Execution of operation op with argument arg on the local copy of process p_i is

indicated by $(local\_obj_i, ret) = execute\_op(local\_obj_i, op, arg)$. $local\_obj_i$ means the local copy held by process p_i. ret is the return value of op after the execution.

Process p_i sends messages to all the other processes using the notation $send_i$.

Process p_i receives a message using the notation $recv_i$.

1. OOP

First we consider the operations belonging to OOP. We want these operations to be executed in each copy in the same order. This order is determined by the timestamps of operations.

Immediately after the operation invocation, we let the invoking process p_j send the operation $op \in OOP$ with its argument and timestamp $\langle local\_time_j, j\rangle$ together to all the other processes. Due to the local clock skew and message delay uncertainty, the order of operation receiving events may be different from the order of operations' timestamps. Each process p_i uses a priority queue $To\_Execute_i$ to temporarily store each $\langle op, arg, ts\rangle$ that it receives and the priority queue uses ts as the key.

$To\_Execute_i$ supports three operations: $add(\langle op, arg, ts\rangle)$, $min()$ and $extract\_min()$. $add(\langle op, arg, ts\rangle)$ inserts $\langle op, arg, ts\rangle$ into $To\_Execute_i$. $min()$ returns the minimum key in this queue. $extract\_min()$ returns the element with minimum key and removes it from the priority queue.

p_i adds an operation into $To\_Execute_i$ immediately after receiving it. p_i does not send messages to itself. So when p_i sends $\langle op, arg, ts\rangle$ to all the other processes, it starts a timer by $set\_timer(d-u, \langle op, arg, ts\rangle, add)$ to count when to add $\langle op, arg, ts\rangle$ into its own priority queue. When this timer expires, it adds $\langle op, arg, ts\rangle$ into $To\_Execute_i$, pretending $\langle op, arg, ts\rangle$ is received through the fastest message, which is delivered $d-u$ time after it is sent.

After op is added into $To\_Execute_i$, p_i waits at most $u+\epsilon$ time before execut-

ing *op* on p_i's local copy. Because due to the message delay bound and local clock skew bound, after holding *op* in $To\_Execute_i$ for at most $u + \epsilon$ time, p_i will not receive any operation whose timestamp is smaller than *op* (formal proof for this point comes after the pseudocode). The waiting time is controlled by another timer, which starts when *op* is added into $To\_Execute_i$ by $set\_timer(u + \epsilon, \langle op, arg, ts\rangle, execute)$. Once $expire\_timer(\langle op, arg, ts\rangle, execute)$ occurs, p_i executes all the operations whose timestamps are no larger than *op* according to their timestamps sequence.

2. MOP and AOP

Now, we think about pure mutators and pure accessors together. We define a parameter X to be a value within $[0, d + \epsilon - u]$. The purpose of this time parameter is to regulate the trade-off of the response times between pure accessors and pure mutators, as in [7]. I.e., if pure accessors respond faster, then pure mutators respond slower, and vice versa.

The pure mutators modify the object. And different execution sequences of several pure mutators may result in different states of the object. For these operations, we still need them to be executed in the same order on each copy, otherwise later the operations may return some illegal value and violate the linearizability of the object. Similarly to the immediately non-self-commuting operations, we give them timestamps with the same mechanism, let their invoking processes send them to all the other processes and use $To\_Execute$ in each process to ensure the same order of their executions on the local copies of the object. Note here the pure mutators and all the other operations except pure accessors share the same $To\_Execute$ in each process, so in the pseudocode, we put the invocation of MOP and OOP together and use *op* to denote an operation which belongs to $MOP \cup OOP$. Because no two operations can be invoked by the same process at the same time, the timestamp of

each operation is unique in $To\_Execute$. So there is no problem to use the timestamps as keys in the priority queue.

However, because pure mutators do not return any information about the object, we can let them respond before they are executed on the local copies of processes. The bottom line is if they do not respond too fast, then the system can distinguish the order of two non-overlapping pure mutators. We let each pure mutator *mop* respond $\epsilon + X$ real time later after the invocation. It guarantees *mop* costs at least ϵ time, which is enough to distinguish the order of two non-overlapping pure mutators. The response time is counted by $set\_timer(\epsilon + X, \langle mop, arg, ts\rangle, respond)$. When this timer expires, the invoking process returns the acknowledgment of the pure mutator.

For the pure accessors, we do not need to broadcast them, because they do not modify the object. And we provide a different mechanism for their timestamps. For the timestamps of pure accesors, we use local invocation times minus X, pretending they are invoked X time earlier. This would not affect the operation results of other concurrent operations, because pure accessors do not modify the object. The benefit is, if a pure accessor *aop* is invoked after a mutator *op* responds, *aop* must have a larger timestamp than *op*. And before *aop* returns, its invoking process executes *op* on the local copy. The execution and response time of *aop* is counted by $set\_timer(d + \epsilon - X, \langle aop, arg, ts\rangle, respond)$. When this timer expires, the invoking process executes all the operations in $To\_Execute$ whose timestamps are smaller than that of *aop* on its local copy of the object, and then execute *aop*.

Since the timestamp of a pure accessor is set to the local clock time minus X, is it possible that this timestamp is equal to the timestamp of another operation? No. In the later analysis, we will show this is impossible and the timestamp of each operation is unique among all the operations.

B. Pseudocode

Now we display the pseudocode of this implementation in Algorithm 1.

Initialization in process p_i: $To\_Execute_i = \emptyset$.

C. Correctness

1. Summary of Observations

Before we prove the correctness of this implementation, we summarize some observations from the pseudocode.

From lines 3-11, we get

Observation C.1 *The earliest real time when each mutator $\langle op, arg, ts\rangle$ is added into any $To\_Execute_i$ (when $set\_timer(u+\epsilon, \langle op, arg, ts\rangle, execute)$ occurs) is $d-u$ time after op is invoked.*

Observation C.2 *The latest real time when each mutator $\langle op, arg, ts\rangle$ is added into any $To\_Execute_i$ (when $set\_timer(u+\epsilon, \langle op, arg, ts\rangle, execute)$ occurs) is d time after op is invoked.*

From lines 12-20, we get

Observation C.3 *When each mutator op is executed by any process p_i, all operations with smaller timestamps in $To\_Execute_i$ have been executed on $local\_obj_i$.*

From lines 21-28, we get

Observation C.4 *When each pure accessor aop responds in p_i, all the operations in $To\_Execute_i$ with smaller timestamps than that of aop have been executed on $local\_obj_i$.*

Algorithm 1 code for process p_i

1: ——————————when $aop(arg) \in AOP$ is invoked——————————
2: $set\_timer(d + \epsilon - X, \langle aop, arg, \langle local\_time_i - X, i\rangle\rangle, respond)$;
3: ——————————when $op(arg) \in MOP \cup OOP$ is invoked——————————
4: **if** $op \in MOP$ **then**
5: $set\_timer(\epsilon + X, \langle op, arg, \langle local\_time_i, i\rangle\rangle, respond)$;
6: **end if**
7: $set\_timer(d - u, \langle op, arg, \langle local\_time_i, i\rangle\rangle, add)$;
8: enable $send_i(\langle op, arg, ts\rangle)$ to all the other processes;
9: ——when $expire\_timer(\langle op, arg, ts\rangle, add)$ OR $recv_i(\langle op, arg, ts\rangle, j), j \neq i$, occurs——
10: $set\_timer(u + \epsilon, \langle op, arg, ts\rangle, execute)$;
11: $To\_Execute_i.add(\langle op, arg, ts\rangle)$;
12: ————when $expire\_timer(\langle op, arg, ts\rangle\, execute)$ occurs——————————
13: **while** $ts > To\_Execute.min()$ **do**
14: $\langle op', arg', ts'\rangle = To\_Execute.extract\_min()$;
15: $(local\_obj_i, ret) = (local\_obj_i, op', arg')$;
16: $cancel\_timer(\langle op', arg', ts'\rangle, execute)$;
17: **if** $ts' = \langle *, i\rangle$ **then**
18: enable return(ret);
19: **end if**
20: **end while**
21: ————when $expire\_timer(\langle aop, arg, ts\rangle, respond)$, $aop \in AOP$ occurs————
22: **while** $ts > To\_Execute.min()$ **do**
23: $\langle op', arg', ts'\rangle = To\_Execute.extract\_min()$;
24: $(local\_obj_i, ret') = (local\_obj_i, op', arg')$;
25: $cancel\_timer(\langle op', arg', ts'\rangle, execute)$;
26: **end while**
27: $(local\_obj_i, ret) = (local\_obj_i, aop, arg)$;
28: enable return(ret);
29: ————when $expire\_timer(\langle mop, arg, ts\rangle, respond)$, $mop \in MOP$ occurs————
30: enable return(ACK);

From lines 4-6 and 29-30, we get

Observation C.5 *Each pure mutator responds $X + \epsilon$ real time after its invocation.*

2. Termination

The termination of pure mutators is trivial from Observation C.5. Now we are going to prove the termination of $oop \in OOP$ and pure accessors.

Lemma C.6 *Each operation $oop \in OOP$ responds at most $d + \epsilon$ real time after its invocation.*

PROOF. Suppose *oop* is invoked by process p_i at real time t. Then $set\_timer(u + \epsilon, \langle oop, arg, ts\rangle, execute)$ occurs at $t + d - u$ and $expire\_timer(\langle oop, arg, ts\rangle, execute)$ occurs at $t + d - u + u + \epsilon = t + d + \epsilon$. At this time, if *oop* hasn't responded, the operations with timestamps smaller than *oop* in $To\_Execute_i$ will be executed one by one. After the executions of them, p_i can execute *oop* on $local\_obj_i$ and then *oop* responds. Since the local execution time is 0, *oop* responds at most $d + \epsilon$ real time later after its invocation. □

Now we prove the termination of each pure accessor.

Lemma C.7 *Each pure accessor aop responds exactly $d + \epsilon - X$ time after its invocation.*

PROOF. Assume *aop* is invoked by process p_i. When $expire\_timer(\langle aop, arg, ts\rangle, respond)$ occurs (which is $d + \epsilon - X$ after the invocation

of *aop*), the only thing that can postpone the response of *aop* is that there exists some operation $\langle op, arg', ts' \rangle$ in $To\_Execute_i$ and $ts' < ts$. At this point, these operations with timestamps smaller than ts will be executed one by one. After all the operations with smaller timestamps in $To\_Execute_i$ are executed, *aop* can be executed on $local\_obj_i$ and then responds. Because the local computation time is 0, *aop* responds exactly $d + \epsilon - X$ time after its invocation. □

3. Linearizability

We show the correctness of the implementation in three steps.

a. Step I

First we prove two important lemmas which will be used later.

Lemma C.8 *After* $expire\_timer(\langle op, arg, ts \rangle, execute)$ *occurs in process* p_i*, no mutators with timestamps smaller than* ts *are added into* $To\_Execute_i$*.*

Proof. Assume, in contradiction that $\langle op', arg', ts' \rangle$ is added into $To\_Execute_i$ after $expire\_timer(\langle op, arg, ts \rangle, execute)$ and $ts' < ts$. Assume *op* is invoked at real time t. By Observation C.1, the earliest time when $set\_timer(u + \epsilon, \langle op, arg, ts \rangle, execute)$ occurs at p_i is $t+(d-u)$ and the earliest time when $expire\_timer(\langle op, arg, ts \rangle, execute)$ occurs at p_i is $t + (d - u) + (u + \epsilon) = t + d + \epsilon$. So $\langle op', arg', ts' \rangle$ must be added into $To\_Execute_i$ after $t+d+\epsilon$. By Observation C.2, op' is invoked after $t+d+\epsilon-d = t+\epsilon$. Because the local clock skew is bounded by ϵ, if op' is invoked more than ϵ time after *op* is invoked, ts' must be larger than ts, contradicting the assumption $ts' < ts$. □

Lemma C.9 *If process p_i invokes a pure accessor aop with timestamp ts in p_i at real time t, no operations with timestamps smaller than ts are added into $To\_Execute_i$ after $t + d + \epsilon - X$ (when $expire\_timer(\langle aop, arg, ts\rangle, respond)$ occurs).*

Proof. Assume, in contradiction that $\langle op', arg', ts'\rangle$ is added into $To\_Execute_i$ after real time $t+d+\epsilon-X$ and $ts' < ts$. Because the local clock skew upper bound is ϵ and since ts uses p_i's local time at real time $t-X$, op' must be invoked no later than $t-X+\epsilon$. Since every message delay is upper bounded by d, p_i receives $\langle op', arg', ts'\rangle$ no later than real time $t-X+\epsilon+d$ and also adds it into $To\_Execute_i$ no later than $t+d+\epsilon-X$, contradicting that op' is added into $To\_Execute_i$ after $t+d+\epsilon-X$. □

b. Step II

Then we use the two lemmas above to prove some lemmas for the real time sequence of different operation types, including mutator after mutator, pure accessor after pure accessor, pure accessor after mutator and mutator after pure accessor.

i. Mutator after Mutator

Lemma C.10 *In every process p_i, all the mutators will be executed on $local\_obj_i$ in the sequence with their timestamps ascending.*

Proof. When p_i executes a mutator op on $local\_obj_i$, by Observation C.3, all the mutators with smaller timestamps in $To\_Execute_i$ have been executed; by Lemma C.8 and Lemma C.9, there is no any other operation with a smaller timestamp received by p_i later. Since we have reliable message delivery between processes, when op is executed, all the operations with smaller timestamps have been received by p_i and executed on $local\_obj_i$. □

Lemma C.11 *A mutator op (invoked by any process p_i) responds at least ϵ time after its invocation.*

PROOF. If *op* is a pure mutator, it responds exactly $\epsilon + X$ after its invocation. Since $X \geq 0$, it responds at least ϵ time after its invocation. If *op* is not a pure mutator, the response time depends on the execution times of the mutators in $To\_Execute_i$. Suppose *op* is invoked at real time t. When *op* responds, there must exist a mutator $\langle op', arg', ts' \rangle$ such that $expire\_timer(\langle op', arg', ts' \rangle, execute)$ occurs and $ts' \geq ts$. By the bounded clock skew, op' is invoked no earlier than $t - \epsilon$. By Observation C.1, $set\_timer(u + \epsilon, \langle op', arg', ts' \rangle, execute)$ occurs no earlier than $t - \epsilon + d - u$ and $expire\_timer(\langle op', arg', ts' \rangle, execute)$ occurs no earlier than $t - \epsilon + d - u + (u + \epsilon) = t + d$. So *op* responds no earlier than $t + d \geq t + \epsilon$. □

Remark: From the proof above, we can answer the question mentioned before: Is it possible that another operation has the same timestamp as a pure accessor *aop*? The answer is NO. Because if such an operation *op* exists, it must be invoked by the same process as *aop*. And we can show:

- $op \notin AOP$. If $op \in AOP$, it must be invoked at the same time by the same process as *aop*, violating that there is only one pending operation in each process at any real time.

- $op \notin MOP$. If $op \in MOP$, since *op* and *aop* have the same timestamp, *op* is invoked X time before the invocation of *aop*. But *op* responds exactly $\epsilon + X > X$ time after its invocation. So when *aop* is invoked, *op* has not responded, violating that there is only one pending operation in each process at any real time.

- $op \notin OOP$. If $op \in OOP$, op responds at least d time after its invocation (from the last sentence in the proof of Lemma C.11, which shows op is invoked at real time t and responds no earlier than $t + d$). Since $d > X$ ($X \leq d + \epsilon - u$ and $\epsilon = (1 - 1/n)u$), similarly to the analysis for MOP, it violates that there is only one pending operation in each process at any real time.

From the analysis above, we can see that op does not exist, and so the answer of that question is NO.

Lemma C.12 *If a mutator op_1 responds before another mutator op_2 is invoked, op_1 is executed before op_2 in each process p_i's local copy $local\_obj_i$.*

Proof. By Lemma C.11, since every mutator costs at least ϵ time to respond, op_2 is invoked more than ϵ time later than op_1 is invoked and so the timestamp of op_2 must be larger than that of op_1. By Lemma C.10, since op_2 has a larger timestamp than that of op_1 and all the mutators are executed with timestamps ascending in each local copy, op_1 is executed before op_2 on $local\_obj_i$. □

ii. Pure Accessor after Pure Accessor

Lemma C.13 *If a pure accessor aop_1 responds in process p_i before another pure accessor aop_2 is invoked in process p_j, all the mutators executed on $local\_obj_i$ before aop_1 returns are executed on $local\_obj_j$ before aop_2 returns.*

Proof. By Lemma C.7, the time between invocation of a pure accessor and its response is $d + \epsilon - X \geq u$ (recall that $0 \leq X \leq d + \epsilon - u$).

Assume in contradiction that there exists a mutator $\langle op, arg, ts \rangle$ which is executed on $local\_obj_i$ before aop_1 responds but executed on $local\_obj_j$ after aop_2 responds. Because all the pure accessors use the same X value, if aop_2 is invoked after

aop_1 returns, it must be invoked at least $d + \epsilon - X \geq u > \epsilon$ time later than aop_1 is invoked (recall we have optimal ϵ, so $u > \epsilon$) and so it has a larger timestamp than aop_1. By Lemma C.9 and Observation C.4, since op is executed on $local\_obj_i$ before aop responds, ts must be larger than the timestamp of aop_2, which is larger than the timestamp of aop_1.

Now we have ts is larger than the timestamp of aop_1 and op is executed before aop_1 returns. So there must exist an operation $\langle op', arg', ts' \rangle$ with $ts' \geq ts$ such that $expire\_timer(\langle op', arg', ts' \rangle, execute)$ occurs before aop_1 returns. Suppose op' is invoked at real time t. By Observation C.1, in process p_i, the earliest time that $set\_timer(u + \epsilon, \langle op', arg', ts' \rangle, execute)$ occurs is $t + d - u$ and the earliest time $expire\_timer(\langle op', arg', ts' \rangle, execute)$ occurs is $t+d+\epsilon$. So aop_1 returns after $t+d+\epsilon$, which means aop_2 returns after $t + d + \epsilon + u$. By Observation C.1, in process p_j, the latest time when $set\_timer(\langle op', arg', ts' \rangle, execute)$ occurs is $t+d$, and the latest time when $expire\_timer(\langle op', arg', ts' \rangle, execute)$ occurs is $t+d+u+\epsilon$, which is before aop_2 responds. Because $ts' \geq ts$, the criteria to execute op on $local\_obj_j$ is reached and so op is executed on $local\_obj_j$ before aop_2 returns, contradicting the assumption about the existence of op. □

iii. Pure Accessor after Mutator

Lemma C.14 *If a pure accessor aop is invoked by p_i after a mutator op responds, p_i executes op on $local\_obj_i$ before aop responds.*

Proof. We prove with two cases as below:

- *op* is a pure mutator.

  Assume *op* is invoked at real time t (local time T). Because *op* costs $\epsilon + X$ time, *aop* must be invoked after $t + \epsilon + X$. Because the local clock skew is

bounded by ϵ, at any real time after $t + \epsilon + X$, the local time of p_i is larger than $(T - \epsilon) + \epsilon + X = T + X$. Then the timestamp for *aop* must be larger than $T + X - X = T$. By Observation C.4 and Lemma C.9, *op* is executed by p_i before *aop* returns.

- *op* is not a pure mutator.

  If *op* is not a pure mutator, the response time depends on the operations in the priority queue $To\_Execute$ of its invoking process.

  Assume *op* is invoked by process p_j and responds at real time t_{res}. There must be a mutator $\langle op', arg', ts' \rangle$ in $To\_Execute_j$ such that $expire\_timer(\langle op', arg', ts' \rangle, execute)$ occurs at t_{res} and $ts' \geq ts$. By the message delay uncertainty u, in process p_i, at real time $t_{res} + u$, $expire\_timer(\langle op', arg', ts' \rangle, execute)$ must have occurred, and so *op* is executed on $local\_obj_i$ no later than $t_{res} + u$. Since $0 \leq X \leq d + \epsilon - u$, *aop* responds after $t_{res} + d + \epsilon - X \geq t_{res} + u$. So p_i executes *op* on $local\_obj_i$ before *aop* responds.

□

iv. Mutator after Pure Accessor

Lemma C.15 *If a mutator op is invoked after a pure accessor aop returns, aop's invoking process will execute op on its local copy after aop returns.*

Proof. *op* can only be executed on the processes' copies after it is invoked and so after the return of *aop*. □

c. Step III

Now we construct a permutation π of any run and then prove π is legal and respects the real time sequence of non-overlapping operations. By Lemma C.10, we already proved that every process executes the mutators in the same order, with timestamps ascending. We arrange all the mutators with their timestamps ascending in π. Then we insert each pure accessor *aop* into the permutation after the latest mutator which is executed on the local copy in *aop*'s invoking process before *aop* responds and before the following mutators.

If there are more than one pure accessors inserted between the mutator op_1 and the mutator op_2, order them by the real times of their invocations, breaking ties with process ids.

Lemma C.16 *If an operation op_1 responds before another operation op_2 is invoked, op_1 appears before op_2 in π.*

Proof. We prove with 4 cases as below:

- op_1 and op_2 are both mutators: By Lemma C.11, op_1 responds at least ϵ time after its invocation. Since op_2 is invoked after op_1 responds, it is invoked more than ϵ time after op_1 is invoked. So op_2 has a larger timestamp than op_1 and is after op_1 in π.

- op_1 and op_2 are both pure accessors: By Lemma C.13, all the mutators executed on the local copy of op_1's invoking process before op_1 responds are executed on the local copy of op_2's invoking process before op_2 responds. Suppose in the construction of π, op_1 is inserted between the mutators op and op'. Then op is executed on the local copy of op_1's invoking process before op_1 responds, so it

is also executed on the local copy of op_2's invoking process before op_2 responds. Therefore op_2 must be inserted into a position after op.

  - op_2 is also inserted between op and op'. Under this condition, since op_1 and op_2 are ordered according to the real times of their invocations, op_1 is before op_2.
  - op_2 is inserted somewhere after op'. Then it is trivial op_2 is after op_1 in π.

- op_1 is a mutator and op_2 is a pure accessor: By Lemma C.14, since op_1 responds before op_2 is invoked, op_1 is executed on the local copy of op_2's invoking process before op_2 responds. Then by the construction of π, op_2 is inserted at some position after op_1.

- op_1 is a pure accessor and op_2 is a mutator: In the construction of π, op_1 must be inserted somewhere before op_2, otherwise op_2 is executed on the local copy of op_1's invoking process before op_1 responds, contradicting Lemma C.15.

□

Lemma C.17 *π is legal.*

PROOF. We prove by induction on the length of π.

Base: π is an empty sequence. There is no operation, so π is vacuously legal.

Induction step: Suppose $\pi = \pi' \circ op$, with the assumption π' is legal, we show π is legal. It is proved in two cases as below:

- $op \in OOP \cup AOP$: By the construction of π (mutators are ordered with timestamps ascending and how pure accessors are inserted) and Lemma C.10, before op is executed on the local copy of its invoking process, op's invoking process has

executed all the mutators in π based on their timestamps ascending sequentially on its local copy, which is exactly the same as the sequence in π. Since the pure accessors in π do not modify the object, it does not make any difference to the execution of *op* if *op*'s invoking process executes them on its local copy or not. Therefore the execution of *op* by *op*'s invoking process on its local copy must return a value such that $\pi \circ op$ is legal.

- $op \in MOP$: *op* is always legal because it does not return any information about the object (refer to Definition D.3).

□

Lemma C.16 and Lemma C.17 prove:

Theorem C.18 *Algorithm 1 is a correct implementation of a linearizable object with an arbitrary data type.*

D. Analysis of Implementations

Theorem D.1 *For all* $mop \in MOP$ *and* $aop \in AOP$, $|mop| + |aop| = d + 2\epsilon$.

When X is 0, the time cost for a pure mutator is ϵ.

With optimal ϵ, the time cost is $(1-1/n)u$, matching the time lower bound (refer to Theorem D.1) in the condition $k = n$. Therefore $(1 - 1/n)u$ is a tight time bound for any pure mutator that is eventually non-self-last-permuting, such as write on a read/write register, push on a stack and enqueue on a queue.

When X is $d + \epsilon - u$, the time cost for a pure accessor is u, leaving a gap $u/2$ to the lower bound [1].

Theorem D.2 *For all* $oop \in OOP$, $|oop| \leq d + \epsilon$.

When $\epsilon \leq u$ and $\epsilon \leq d/3$, this upper bound matches the lower bound for immediately non-self-commuting operations (refer to Theorem C.1).

CHAPTER VI

TIME BOUNDS FOR SPECIFIC SHARED OBJECTS

In this chapter, we summarize the new lower bounds and upper bounds for specific objects, and compare them with previous results.

A. Operations on Read/Write/Read-Modify-Write Registers

There are three operations on a Read/Write/Read-Modify-Write Register: read, write and read-modify-write. Read is a pure accessor. Write is a pure mutator. Read-modify-write is an immediately non-self-commuting operation. With the implementation in Chapter V (Algorithm 1), the read operation costs $d + \epsilon - X$ time and the write operation costs $\epsilon + X$, where X is a variable within $[0, d + \epsilon - u]$. Then the write operation costs ϵ when X is set to 0 and so the write operation costs $(1 - 1/n)u$ when ϵ is optimal, which matches the time lower bound for pure mutators (Theorem D.1). Besides, the read-modify-write operation costs $d + \epsilon$ in the implementation, partially matching the time lower bound for immediately non-self-commuting operations (Theorem C.1). The most significant gap between time upper bound and lower bound is the time cost for $|write| + |read|$. The gap is 2ϵ, which actually depends on the message delay uncertainty. The time bounds are shown as Table I.

B. Operations on Queues and Stacks

The operations on a queue and those on a stack are very similar. Peek is a pure accessor. Enqueue and push are pure mutators. Dequeue and pop are immediately non-self-commuting operations. The time complexities of enqueue and push operations are exactly the same as for the write operation on a register, whose tight time

Table I. Summary of Operation Time Bounds on Read/Write/Read-Modify-Write Register

operations	Previous Lower Bound	Lower Bound	Upper Bound
read-modify-write	d [3]	$d + \min\{\epsilon, u, d/3\}$	$d + \epsilon$
write	$u/2$ [1]	$(1 - 1/n)u$	ϵ
read	$u/2$ [3]	-	u
write + read	d [5]	d	$d + 2\epsilon$

bound is $(1 - 1/n)u$. The time complexity of dequeue and pop operations are exactly the same as for the read-modify-write on a register, whose tight time bound is $d + \epsilon$ with $\epsilon \leq u$ and $\epsilon \leq d/3$. The difference from the analysis on a read/write/read-modify-write register is the time complexity of $(|push| \text{ or } |enqueue|) + |peek|$. Because the push operation and the enqueue operation do not overwrite the whole state of the object, the lower bound for $(|push| \text{ or } |enqueue|) + |peek|$ is $d + \epsilon$ while the lower bound for $|write| + |read|$ is d. The time bounds are shown as Table II and Table III.

Table II. Summary of Operation Time Bounds on Queue

operations	Previous Lower Bound	Lower Bound	Upper Bound
enqueue	$u/2$ [1]	$(1 - 1/n)u$	ϵ
dequeue	d [3]	$d + \min\{\epsilon, u, d/3\}$	$d + \epsilon$
enqueue + peek	d [3]	$d + \min\{\epsilon, u, d/3\}$	$d + 2\epsilon$

C. Operations on Trees

We consider four operations on a rooted tree: insert, delete, search and depth. Insert and delete are pure mutators. Search and depth are pure accessor. There is no

Table III. Summary of Operation Time Bounds on Stack

operations	Previous Lower Bound	Lower Bound	Upper Bound
push	$u/2$ [1]	$(1-1/n)u$	ϵ
pop	d [3]	$d+\min\{\epsilon, u, d/3\}$	$d+\epsilon$
push + peek	d [3]	$d+\min\{\epsilon, u, d/3\}$	$d+2\epsilon$

operation which is both mutator and accessor. The time bounds result is summarized as Table IV:

Table IV. Conclusions of Operation Time Bounds on Tree

operations	Previous Lower Bound	Lower Bound	Upper Bound
insert	$u/2$ [3]	$(1-1/n)u$	ϵ
delete	$u/2$ [3]	$(1-1/n)u$	ϵ
insert + depth	d [3]	$d+\min\{\epsilon, u, d/3\}$	$d+2\epsilon$
delete + depth	d [3]	$d+\min\{\epsilon, u, d/3\}$	$d+2\epsilon$

CHAPTER VII

CONCLUSION

In this work, we studied the properties of operations on an arbitrary object with linearizability. Then we showed the time lower bounds and upper bounds for three categories of operation types: immediately non-self-commuting operations, eventually non-self-commuting operations and immediately non-commuting pair of operations. In each category, we present our results on time bounds and discuss the remaining future work:

The time lower bound for immediately non-self-commuting operation types is $d + \min\{\epsilon, u, d/3\}$ and the time bound is tight if $\epsilon \leq d/3$ (d is message delay upper bound, u is message delay uncertainty, ϵ is local clock skew upper bound). The tight time bound for the case $\epsilon > d/3$ is still unknown. Considering the optimal $\epsilon = (1 - 1/n)u$, if we have $u \leq d/3$, it is easy to get $\epsilon \leq d/3$. So the challenge comes from the message uncertainty u, if it is larger than $d/3$. From an intuitive view, with a larger message uncertainty, each process can get less information (about when the message is sent) from time when the message is received, and therefore it is harder to decide the sequence of concurrent operations.

For the operation types of which there exist k instances, such that different permutations of them are not equivalent, the lower bound is $(1-1/k)u$. For eventually non-self-last-permuting operation types, $k = n$, and so the lower bound is $(1 - 1/n)u$ and it is a tight time bound when the local clock skew upper bound is optimal. Although this result can be applied to many commonly used objects such as registers, queues and stacks, it remains an open question about the tight time lower bound for operation types which are eventually non-self-commuting but not eventually non-self-last-permuting. Since there are $k!$ possible permutations for k concurrent operations,

and we have studied the case where two permutations have different last operations, it is a challenge to study all the other cases.

The time lower bound for a pair of the non-commuting operation types OP and AOP depends on the properties of OP. If OP is eventually self-commuting, such as insert and delete on a set, or if OP is eventually non-self-commuting but overwriting, such as the write operation on a register, the lower bound is d, there remains a gap of 2ϵ with the upper bound, according to our implementation where $|MOP| + |AOP| = d + 2\epsilon$. If OP is eventually non-self-commuting and non-overwriting, such as enqueue on a queue and push on a stack, the lower bound is $d + \epsilon$. We narrowed the gap between upper and lower bound, but still leave a gap of ϵ.

In this work, we assumed bounded local clock skew without time drift. The partially synchronous model with bounded clock skew and bounded time drift needs to be explored in the future works. We may also consider different types of failures in message passing systems.

REFERENCES

[1] H. Attiya and J.L. Welch, "Sequential consistency versus linearizability," *ACM Trans. Computer Syst.*, vol. 12, pp. 91–122, May 1994.

[2] M.P. Herlihy and J.M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Languages Syst.*, vol. 12, pp. 463–492, July 1990.

[3] M.J. Kosa, "Time bounds for strong and hybrid consistency for arbitrary abstract data types," *Chicago Journal of Theoretical Computer Science*, 1999.

[4] L. Lamport, "On Interprocess Communication," *Distributed Computing*, vol. 1, no. 2, pp. 86–101, 1986.

[5] R.J. Lipton and J.S. Sandberg, "PRAM: A scalable shared memory," Technique Report CS-TR-180-88, Princeton Univ., Dept. of Computer Science, 1988.

[6] J. Lundelius and N. Lynch, "An upper and lower bound for clock synchronization," *Information and Control*, vol. 62, no. 2–3, pp. 190–204, 1984.

[7] M. Mavronicolas and D. Roth, "Linearizable read/write objects," in *Theoretical Computer Science*, vol. 220, no. 1, pp. 267–319, June 1999.

[8] W.E. Weihl, "Commutativity-based concurrency control for abstract data types," in *IEEE Trans. Computers*, vol. 37, no. 12, pp. 1488–1505, 1988.

VITA

Name: JIAQI (Erica) WANG

E-mail: ericaqi@cse.tamu.edu Mobile: (979) 393-8528

Address: Department of Computer Science and Engineering

Texas A&M University

TAMU 3112, c/o Jennifer Welch

College Station, TX 77843-3112

EDUCATION

Aug 2008-Dec 2011 in Texas A&M University, College Station, Texas

Master of Science in Computer Science and Engineering

Sep 2001-Jun 2005 in Nanjing University, Nanjing, China

Bachelor of Science in Computer Science

COURSE PROJECT EXPERIENCE

Aug 2008-Present Computer Science & Engineering, Texas A&M University

Tiny-SQL Interpreter, Container Loading Recommendation, SOM Clustering, Parallel Hierarchical Clustering, Trust Propagation with Controversial User and Online Mini Texas Hold'em

RESEARCH EXPERIENCE

Aug 2008-May 2009 in Industrial Engineering, Texas A&M University

Project: Intelligent Tutoring System for Automatic Assembly System

Jan 2010-Present in Computer Science &Engineering, Texas A&M University

Project: Distributed Algorithms Design and Analysis

PUBLICATION

Brief Announcement: Time Bounds for Shared Objects in Partially Synchronous Systems, PODC'11